OBJECT WINDOWS® C++

PROGRAMMIER-HANDBUCH

BORLAND

ObjectWindows für C++

ObjectWindows für C++

Programmierhandbuch

BORLAND

Copyright © 1991 by Borland International, INC.
Jede Vervielfältigung dieses Handbuches sowie des ObjectWindows Softwareprogrammes wird strafrechtlich verfolgt. Die Rechte an der Dokumentation und die Rechte an dem Softwareprogramm ObjectWindows liegen bei Borland International.

Der rechtmäßige Erwerb der Programmdisketten und der Handbücher erlaubt die Nutzung der Programme analog der Benutzung eines Buches. Entsprechend der Unmöglichkeit, daß ein Buch zugleich an verschiedenen Orten von mehreren Personen gelesen wird, darf das Softwareprogramm ObjectWindows nicht gleichzeitig von verschiedenen Personen an verschiedenen Orten und auf verschiedenen Geräten benutzt werden. Diskettenkopien dürfen lediglich zum Zweck der Datensicherung angefertigt werden.

Sie (der Lizenznehmer von ObjectWindows) dürfen eigene Programme, die Sie mit dem Borland C++-Sprachcompiler geschrieben und compiliert haben, ohne Einschränkung oder Verpfichtung zu zusätzlichen Lizenzgebühren verwenden, weitergeben oder verkaufen, solange diese Programme einen Copyright-Vermerk enthalten. Unter Copyright-Vermerk verstehen wir entweder Ihren eigenen Copyright-Vermerk oder gegebenenfalls einen Hinweis wie »Created using ObjectWindows by Borland International 1991«.

Die Beispielprogramme von ObjectWindows, die auf den Originaldisketten mitgeliefert werden, demonstrieren die verschiedenen Eigenschaften von Obejct-Windows. Sie dienen ausschließlich zu Lehrzwecken. Sie dürfen diese Beispielprogramme editieren oder für den eigenen Gebrauch modifizieren. Sie dürfen diese Programme jedoch nicht als Ganzes oder in Teilen in compilierter Form oder als Quelltext weitergeben oder verkaufen. Sie dürfen die Routinen aus den Beispielprogrammen in eigenen Anwendungsprogrammen verwenden, solange die resultierenden Programme nicht einem Bespielprogramm oder einem Teil eines Beispielprogrammes in Erscheinung oder Funktion gleichen.

Einschränkung der Gewährleistung
Es wird keine Garantie für die Richtigkeit des Inhaltes dieses Handbuches übernommen. Da sich Fehler, trotz aller Bemühungen, nie vollständig vermeiden lassen, sind wir für Hinweise jederzeit dankbar.

Herausgeber
Borland GmbH
Gautinger Straße 10
Postfach
8130 Starnberg
Telefon 0 81 51 / 2 63 - 0
Telefax 0 81 51 / 149 09

Umschlag und typografische Konzeption: Rudolf Paulus Gorbach

Übertragung ins Deutsche: Peter Baier, Roland Konstanty, Waltraud Kopelent

Typographische Konzeption des Innenteils: Rudolf Paulus Gorbach

Grafiken: Sid Gastl

Satz: Hanna Bergbauer

PostScript-Belichtung: CD Computer & Dokumentations GmbH, Aalen

Druck und Bindung: Kösel GmbH, Kempten

2., unveränderte Auflage, Mai 1992

Inhalt

Einführung 1

Das ObjectWindows-Paket 2
Der Inhalt dieses Handbuchs 2
Hard- und Software-Voraussetzungen 3
ObjectWindows installieren 3
INSTALL verwenden 3
Die Datei README 4
Die Hilfedateien 4
Hilfe unter Windows 4
Hilfe unter DOS 5
Die Typographie in diesem Handbuch 6
Namenskonventionen in ObjectWindows 6
Fragen und Probleme – die Beratung der Borland GmbH 7

I. Eine Einführung in ObjectWindows 9

1. Das Fenster vererben 11

Was ist eine Windows-Anwendung? 11
Vorteile von Windows 12
Anforderungen an die Hardware 12
Objektorientierte Programmierung für Windows 13
Eine bessere Schnittstelle zu Windows 13
Die Struktur einer Windows-Anwendung 18
Die Struktur von Windows 19
Interaktion mit Windows und DOS 19
»Hello, Windows« 20
Vorgehensweise bei der Anwendungsentwicklung 22

2. Schritt für Schritt durch Windows 23

Aufbau einer ObjectWindows-Anwendung: Vorbereitungen 24
Die Container-Klassenbibliothek 24

Verzeichnisse 24
Die korrekte Bibliothek festlegen 25
Die Ressourcendatei erstellen 26
Aufbau einer ObjectWindows-Anwendung: die Einzelheiten 27
Die IDE verwenden 27
Den Kommandozeilencompiler verwenden 29
Schritt 1: Eine einfache Windows-Anwendung 30
Voraussetzungen 30
Schritt 2: Die Klasse für das Hauptfenster 33
Was ist ein Fensterobjekt? 33
Eine neue Fensterklasse erstellen 34
Botschaften beantworten 35
Ein Programm beenden 37

3. Ein Fenster füllen 41

Was ist ein Bildschirmkontext? 41
Schritt 3: Text in einem Fenster zeichnen 42
Botschafts-Struktur 43
Den Bildschirm löschen 44
Schritt 4: Linien in einem Fenster zeichnen 44
Bewegungen der Maus 45
Mausbewegungs-Botschaften beantworten 46
Schritt 5: Die Linienstärke ändern 48
Einen neuen Stift auswählen 48
Ein Eingabefeld verwenden 51
Schritt 6: Grafiken zeichnen 52
Das Zeichen-Modell 52
Grafiken als Objekte speichern 53
Gespeicherte Grafiken neu zeichnen 55

4. Ein Menü hinzufügen 57

Menüressourcen 57
Schritt 7: Ein Menü für das Hauptfenster 59
Eine Menübotschaft abfangen 59
Eine Menübotschaft beantworten 60

5. Einen Dialog erstellen 63

Schritt 8: Ein Pop-up-Fenster hinzufügen 64
Ein Pop-up-Fenster erzeugen und anzeigen 64

Die Funktion MakeWindow 65
Ein Dialogfenster hinzufügen 66
Ein Datenelement hinzufügen 67
Einen Dialog führen 67
Schritt 9: Die Zeichnung in einer Datei speichern 68
Den Status überwachen 69
Dateien speichern und laden 71

6. Fenster anzeigen 73

Module mit ObjectWindows verwenden 74
Das Hauptprogramm ändern 74
Ein Modul erzeugen 74
Steuerelemente in einem Fenster 76
Was sind Steuerelemente? 77
Steuerelemente für Fenster erstellen 77
Steuerelementobjekte als Datenelemente 78
Steuerelemente verwalten 78
Steuerelement-Ereignisse beantworten 80

II. ObjectWindows verwenden 83

7. Überblick über ObjectWindows 85

ObjectWindows-Konventionen 85
Die ObjectWindows-Hierarchie 86
Object 86
TModule 86
TApplication 86
Schnittstellenobjekte 87
Fensterobjekte 88
Dialogobjekte 88
Steuerelementobjekte als Datenelemente 89
MDI-Objekte 90
Scroller-Objekte 90
Windows-Funktionen 91
ObjectWindows-Aufrufe von Windows-Funktionen 91
Zugriff auf Funktionen 91
Stilkonstanten kombinieren 92
Typen von Windows-Funktionen 93
Callback-Funktionen 93

Windows-Botschaften 95
Die Parameter einer Windows-Botschaft 96
Typen von Windows-Botschaften 97
Standardbearbeitung von Botschaften 99
Botschaften senden 99
Botschaftsbereiche 100
Anwenderdefinierte Botschaften 101

8. Anwendungsobjekte und -Module 103

Den Anwendungsfluß steuern 103
Anwendungen initialisieren 104
Das Hauptfenster initialisieren 104
Eine Anwendungsinstanz initialisieren 106
Die erste Anwendungsinstanz initialisieren 107
Anwendungen ausführen 108
Anwendungen schließen 109

9. Schnittstellenobjekte 111

TWindowsObject 111
Wofür Schnittstellenobjekte benötigt werden 111
Das Verhältnis zwischen über- und untergeordneten Fenstern 113
Listen untergeordneter Fenster 113
Über untergeordnete Fenster iterieren 114
Botschaftsbearbeitung 115
Botschaften beantworten 115
Befehlsabhängige Botschaften und Botschaften untergeordneter Fenster 116
Botschaftsbearbeitung durch Standardantworten 118

10. Fensterobjekte 121

Die Klasse TWindow 121
Fensterobjekte initialisieren und erstellen 122
Ein Fensterobjekt initialisieren 122
Fensterelemente erstellen 125
Zusammenfassung 125
Windows-Fensterklassen-Registrierung 126
Registrierungsattribute 127
Bildlauf 129
Scroller-Attribute 130

Dem Fenster einen Scroller zuweisen 131
Ein Bildlauf-Beispiel 131
Automatischer Bildlauf und Nachführung 133
Die Schrittweiten und den Bildlaufbereich ändern 134
Die Bildlaufposition ändern 134
Die Seitengröße bestimmen 135
Zeichenelementfunktionen für den Bildlauf optimieren 135
Editier- und Dateifenster 136
Editierfenster 136
Dateifenster 139

11. Dialogobjekte 141

Dialogressourcen verwenden 141
Untergeordnete Dialogobjekte verwenden 142
Dialoge erzeugen 142
Dialoge erstellen und ausführen 142
Einen untergeordneten Dialog beenden 144
Einen Dialog als Hauptfenster verwenden 144
Steuerelemente manipulieren und Botschaften bearbeiten 145
Erweitertes Beispiel für einen Dialog 149
Eingabedialoge 149
Datei-Dialoge 151

12. Steuerelementobjekte 153

Steuerelementobjekte verwenden 154
Steuerelementobjekte initialisieren und erstellen 155
Steuerelementobjekte freigeben und entfernen 155
Steuerelemente und Botschaftsbearbeitung 156
Benachrichtigungsbotschaften der Steuerelemente beantworten 156
Steuerelement-Fokus und die Tastatur 158
Listen 158
Listen initialisieren und erstellen 158
Listen ändern 159
Listen abfragen 160
Aus einer Liste wählen 161
Kombinationsfenster 162
Drei Arten von Kombinationsfenstern 163
Einen Kombinationsfenstertyp auswählen 164
Kombinationsfenster erzeugen 164
Kombinationsfenster ändern 165
Beispielprogramm: CBoxTest 165

Statische Steuerelemente 165
Statische Steuerelemente erzeugen 166
Informationen über statische Steuerelemente 167
Statische Steuerelemente ändern 167
Beispiel: Die Anwendung StatTest 167
Editierfelder 167
Editierfelder erzeugen 168
Zwischenablage und Editieroperationen 170
Informationen über Editierfelder 170
Editierfelder ändern 172
Beispielprogramm: EditTest 174
Aktionsschalter-Steuerelemente 174
Benachrichtigungsbotschaften beantworten 174
Auswahl- und Schaltfelder 175
Auswahl- und Schaltfelder erzeugen 175
Informationen über den Status von Wahlfeldern 176
Wahlfeld-Status ändern 176
Auswahl- und Schaltfeldbotschaften beantworten 177
Gruppen 177
Gruppen erzeugen 178
Gruppenbotschaften beantworten 178
Beispielprogramm: BtnTest 178
Bildlaufleisten 179
Bildlaufleisten erzeugen 180
Informationen über Bildlaufleisten 181
Bildlaufleisten ändern 181
Bildlaufereignisse beantworten 182
Beispiel: SBarTest 184
Steuerelement-Daten übertragen 184
Einen Transferpuffer definieren 184
Steuerelemente erzeugen und den Transfer aktivieren 186
Daten übertragen 187
Den Transfer für Steuerelemente anpassen 188
Transfer-Beispiel 188

13. Anwenderdefinierte Steuerelementobjekte 191

Ein voreingestelltes Steuerelement modifizieren 191
Die Erstellungsstile ändern 191
Ein voreingestelltes Steuerelement als zeichenbar definieren 192
Standard-Botschaftsantworten ändern 193
Verwendung eines anwenderdefinierten Steuerelements 194
Ein Steuerelement entwerfen 194

14. MDI-Objekte 197

Die Komponenten einer MDI-Anwendung 197
Das MDI-Fenster – TWindow und TMDIClient 198
MDI-Fenster erzeugen 198
MDI-Rahmenfenster erzeugen 199
Untergeordnete MDI-Fenster erzeugen 199
Botschaften in MDI-Anwendungen bearbeiten 200
Untergeordnete MDI-Fenster bearbeiten 201
Aktivierung von untergeordneten Fenstern 201
Das Menü für untergeordnete Fenster 201
Beispiel für eine MDI-Anwendung 202

15. Streams und Objekte 203

Die Bibliothek iostream 204
Die überschriebenen Operatoren << und >> 205
Stream-fähige Klassen und TStreamable 206
Die Stream-Verwaltung 207
Konstruktoren stream-fähiger Klassen 209
Namen für stream-fähige Klassen 210
Verwendung der Stream-Verwaltung 210
Den Code der Stream-Verwaltung linken 211
Ein Stream-Objekt erstellen 211
Das Stream-Objekt verwenden 212
Kollektionen und Streams 212
Array als stream-fähig definieren 212
Die stream-fähige Klasse TPointArray 213
Die Elementfunktion streamableName 214
Die stream-fähige Funktion read 215

III. Windows programmieren 217

16. Speicherverwaltung 219

Die Speicherverwaltung von Windows 219
Die Verwaltung des Speichers durch Windows 220
Handles und Adressen 221
Lokaler und globaler Speicher 221
Der lokale Heap 222
Lokalen Speicher reservieren und verwenden 223

Lokale Speicherblöcke freigeben und verwerfen 224
Lokale Speicherblöcke neureservieren und ändern 225
Attribute lokaler Speicherblöcke abfragen 225
Hinweise für lokale Speicherblöcke 226
Der globale Heap 226
Reservierung und Verwendung des globalen Speichers 226
Weitere Möglichkeiten zum globalen Speichermanagement 228
Globale Speicherblöcke reservieren und verwenden 228
Globale Speicherblöcke freigeben und verwerfen 229
Globale Speicherblöcke neureservieren und ändern 229
Globale Speicherblöcke abfragen 229
Globales Verwerfen 230
Speicher-Attribute 230
Hinweise für den globalen Speicher 230

17. Dynamische Linkbibliotheken 233

DLLs in C++ 233
DLL-Funktionen schreiben 233
DLL-Funktionen exportieren 235
DLL-Funktionen aufrufen 235
ObjectWindows-DLLs 236
Gemeinsam genutzte DLLs 237
Eine ObjectWindows-DLL aus einer Nicht-ObjectWindows-Anwendung aufrufen 242
ObjectWindows als DLL 243

18. Dynamischer Datenaustausch 245

Die Zwischenablage von Windows 245
Formate für die Zwischenablage 245
Daten in der Zwischenablage ablegen 246
Daten aus der Zwischenablage abrufen 247
Botschaften zwischen Anwendungen übermitteln 249
Dynamischer Datenaustausch 249
Begriffe 250
Aufbau einer DDE-Konversation 251
Beenden einer DDE-Konversation 253
Elementfunktionen für DDE 254
Anfordern eines DDE-Elements 254
Kontinuierlicher Datentransfer 257
Den Server zum Ändern eines Elements auffordern 258
Makrobefehle im Server ausführen 259

Das Thema »System« 260

19. Die grafische Geräteschnittstelle 261

Bildschirmkontexte 262
Der Inhalt des Bildschirmkontexts 262
Vordefinierte Zeichenwerkzeuge 264
Virtuelle Zeichenwerkzeuge 265
Grafikanzeige in Fenstern 271
Fenster-Strategie 272
Grafik-Strategie 272
Zeichenwerkzeuge 273
GDI-Zeichenfunktionen 275
Funktionen zum Zeichnen von geometrischen Formen 278
Der Einsatz von Paletten 282
Erstellen einer Palette 282
Zeichnen mit einer Palette 282
Informationen über Paletten 283
Ändern einer Palette 283
Palettenbotschaften 284

20. Details über Ressourcen 287

Ressourcen erstellen 287
Ressourcen-Script-Dateien 289
Hinweise für die Benutzung des Ressourcencompilers 292
Menüs laden 293
Tastenkürzel-Tabellen laden 294
Dialogfenster laden 295
Mauszeiger und Symbole laden 295
Ressourcen in einer Anwendung laden 296
String-Ressourcen laden 296
Bitmaps laden 297
Einen Pinsel mit Hilfe eines Bitmaps erstellen 298
Bitmaps in Fenstern anzeigen 300

21. Richtlinien für Standardanwendungen 301

Gestaltungsprinzipien 301
Antworten, die der Erwartung des Anwenders entsprechen 301
Steuerung des Programm-flusses durch den Anwender 302
Standards für Erscheinungsbild und Verhalten 303

Generelle Präsentation 303
Maus- und Tastatur-Interaktionen 303
Menüs 304
Weitere Anmerkungen zur Gestaltung 305
Sichere Programme 306
Programmieren unter dem Motto »Alles oder Nichts« 306
Weitere Fehler bei der Erstellung von Fenstern 308
Großverbraucher 309

IV. ObjectWindows-Referenz 311

22. ObjectWindows-Klassen 313

23. Stream-fähige Klassen in ObjectWindows 411

24. Weitere Komponenten von ObjectWindows 427

Einführung

ObjectWindows eröffnet Ihnen neue Wege für die Entwicklung von Windows-Anwendungen. Bisher wurde zur Programmierung fast ausschließlich die Sprache C und eine Vielzahl von separaten, komplexen Programmiertools verwendet. Das Resultat war ein langwieriger, komplizierter und sogar verwirrender Programmierprozeß. Mit ObjectWindows ist die Windows-Programmierung spürbar einfacher geworden.

Bei der Programmierung für Windows werden Sie mit einigen Neuheiten konfrontiert. Dazu gehört zum Beispiel die Arbeit mit Text und Grafiken in größenvariablen Fenstern, die Interaktion mit anderen Programmen in einer Multitasking-Umgebung und die Verwendung von fast 600 Funktionen der Windows-API. Der aufwendige Teil der Arbeit liegt darin, sicherzustellen, daß die grundlegenden Funktionen des Programms auch unter Windows korrekt ablaufen – und daß dies bei allen Funktionen der Fall ist.

Mit ObjectWindows können Sie all dies ausführen. ObjectWindows ist eine objektorientierte Bibliothek, die das Verhalten und die Eigenschaften (auf der Anwendungs- und Fensterseite) enthält, die Windows-Anwendungen normalerweise ausführen.

ObjectWindows erleichtert die Windows-Programmierung aufgrund folgender Eigenschaften:

- ObjectWindows ist eine konsistente, intuitive und vereinfachte Schnittstelle zu Windows.
- ObjectWindows stellt das Verhalten und die Eigenschaften für die Fensterverwaltung und Botschaftsbearbeitung zur Verfügung.
- ObjectWindows ist ein Grundgerüst für die Strukturierung einer Windows-Anwendung.

Sie »erben« diese Grundfunktionalität und können sich deshalb auf die Erfordernisse konzentrieren, die für Ihre spezielle Anwendung zu erfüllen sind.

Die ObjectWindows-Klassen beinhalten nicht alle logischen Gegebenheiten einer Windows-Anwendung. Bei der wiederholten Verwendung der von ObjectWindows zur Verfügung gestellten Klassen können Sie die Entwicklungszeit für Ihre Anwendung spürbar verkürzen. Außerdem müssen Sie weitaus weniger Quelltext verwalten und pflegen.

Das ObjectWindows-Paket

Zum Lieferumfang von ObjectWindows gehören die Originaldisketten und zwei Handbücher: Das *Programmierhandbuch* und das *Windows-Referenzhandbuch*

Die Disketten enthalten den ObjectWindows-Code, den Sie zu Beginn Ihrer Programmierung von Windows-Anwendungen unter dem objektorientierten Aspekt von ObjectWindows benötigen. Sie finden auf diesen Disketten alle Beispielprogramme, auf die in diesem Handbuch Bezug genommen wird, und weitere Programme, die Sie sich ansehen und aus denen Sie lernen können.

Hinweis: Auf den Disketten finden Sie auch compilierte ObjectWindowsBibliotheken, die Ihnen Arbeit ersparen.

Je nachdem, mit welchem Borland-Produkt Sie ObjectWindows erworben haben, stehen Ihnen folgende Hilfedateien zur Verfügung:

- OWLWHELP.HLP für die Windows Hilfeanwendung;
- OWLTHELP.OWH für DOS THELP.

Der Inhalt dieses Handbuchs

Da ObjectWindows einige neue Techniken verwendet, enthält das vorliegende Handbuch sehr viele Beispiele und eine vollständige Referenz zu ObjectWindows. Es gliedert sich in vier große Abschnitte:

Teil 1 »Eine Einführung in ObjectWindows« macht Sie mit den Konzepten der Windows-Programmierung vertraut. Hier finden Sie auch ein Tutorial, das Sie Schritt für Schritt in die Entwicklung und Erweiterung einer ObjectWindows-Anwendung einführt.

Teil 2 »Mit ObjectWindows arbeiten« geht detailliert auf die Elemente von ObjectWindows ein. Sie finden hier einen Überblick über die Objekthierarchie und ihre Wechselbeziehung mit der Windows-Umgebung. Die Bestandteile der Hierarchie und ihr Einsatz werden ausführlich erläutert.

Teil 3 »Unter Windows programmieren« gibt einen Überblick über die Aspekte der Windows-Programmierung, die in ObjectWindows nicht zur Verfügung stehen. Dazu gehören Grafiken, dynamische Linkbibliotheken, die Kommunikation zwischen Anwendungen, der Gebrauch von Ressourcen, Speicherverwaltung, die gemeinsame Nutzung von Code und Daten. Dieser Handbuchteil enthält außerdem grundsätzliche Regeln, die bei der Windows-Programmierung beachtet werden müssen.

Teil 4 »ObjectWindows-Referenz« listet die in der ObjectWindows-Bibliothek enthaltenen Klassen, Typen und Konstanten auf. Dabei werden alle Elementfunktionen, Datenelemente und friend-Funktionen detailliert erläutert.

Das *Windows-Referenzhandbuch* enthält eine Referenz aller Windows-Funktionen, -Typen und -Konstanten. Das gesamte Material ist alphabetisch gegliedert, so daß Sie die gewünschten Einträge schnell und einfach auffinden können.

Hard- und Software-Voraussetzungen

Da ObjectWindows ein Anwendungsrahmen für die Programmierung von Windows-Anwendungen ist, müssen für ObjectWindows-Anwendungen dieselben Hardware-Bedingungen erfüllt werden wie für Windows-Anwendungen:

- Festplatte
- 2 MByte Speicher oder mehr
- Windows-kompatible Grafikkarte
- Windows 3.0 oder höher im Standard- oder erweiterten 386-Modus.

Sie können ObjectWindows-Anwendungen (und den ObjectWindows-Quelltext selbst) mit jedem Borland-C++-Compiler compilieren, der Windows-Anwendungen erstellen kann.

ObjectWindows installieren

ObjectWindows wird mit einem Installationsprogramm namens INSTALL ausgeliefert. Da die Dateien in einem komprimierten Format vorliegen, müssen Sie dieses Programm verwenden. Sie können die ObjectWindows-Dateien nicht einfach auf die Festplatte kopieren. INSTALL kopiert und dekomprimiert diese Dateien. In der Datei FILELIST.DOC sind alle mitgelieferten Dateien aufgeführt.

Fertigen Sie – bevor Sie mit Ihrer Arbeit beginnen – eine komplette Arbeitskopie der Originaldisketten an und verwahren Sie letztere an einem geeigneten Ort. Lesen Sie die Nutzungsvereinbarung, die dem Programmpaket beiliegt, sorgfältig durch und schicken Sie uns auf jeden Fall die ausgefüllte Registrierungskarte zurück. Nur so können wir Sie immer über die neuesten Upgrades und Versionen von ObjectWindows unterrichten.

INSTALL verwenden

INSTALL legt unter anderem Verzeichnisse an, die für das Speichern der ObjectWindows-Dateien erforderlich sind, und überträgt die Dateien von den Originaldisketten auf die Festplatte. Die dabei ausgeführten Arbeitsschritte sind selbsterklärend. Sie müssen nur folgendes wissen:

1. Legen Sie die Installationsdiskette (Diskette 1) in Laufwerk A. Geben Sie folgenden Befehl ein und drücken Sie danach CR:

```
A:INSTALL
```

2. Drücken Sie im Installationsbildschirm CR.

3. Führen Sie die Anweisungen aus, die auf dem Bildschirm eingeblendet werden.

Die Datei README

Die Datei README enthält Informationen über Änderungen, die nach der Drucklegung des Handbuches hinzugekommen sind. Auf diese Datei können Sie folgendermaßen zugreifen:

1. Falls ObjectWindows noch nicht installiert ist, legen Sie die Installationsdiskette in Laufwerk A. Ist ObjectWindows bereits installiert, führen Sie Arbeitsschritt 3 aus.

2. Geben Sie `A:` ein und drücken Sie CR.

3. Geben Sie `README` ein und drücken Sie CR. Mit den Pfeiltasten UP und DOWN können Sie durch die angezeigte Datei blättern.

4. Mit Esc wird die Datei verlassen.

Die Hilfedateien

Es steht Ihnen eine der folgenden ObjectWindows-Hilfedateien zur Verfügung: OWLWHELP.HLP (Hilfedatei unter Windows) und OWLTHELP.OWH (Hilfedatei unter DOS). Die Hilfeinformationen sind in beiden Dateien identisch, der einzige Unterschied besteht in der Formatierung. Welche dieser Dateien Sie haben, hängt vom Produkt ab, mit dem Sie ObjectWindows erworben haben.

Hilfe unter Windows

OWLWHWLP.HLP ist eine Hilfedatei für die Windows-Hilfeanwendung WINHELP.EXE. Sie könnten sich beispielsweise ein Symbol für den Programm-Manager mit folgenden Eigenschaften erstellen:

- Beschreibung: ObjectWindows Hilfe
- Kommandozeile: `WINHELP.EXE C:\BORLANDC\OWL\OWLWHELP.HLP`

Klicken Sie dieses Symbol zweimal an, um OWLWHELP.HLP unter WINHELP.EXE zu laden. Wenn Sie ObjectWindows nicht in den Standardverzeichnissen installiert haben, müssen Sie den Pfadnamen für OWLWHELP.HLP ändern.

Hilfe unter DOS

THELP ist ein speicherplatzresidentes Zusatzprogramm, das Online-Hilfe für ObjectWindows unter DOS anbietet. Wenn Sie Ihre ObjectWindows-Anwendung beispielsweise in der IDE von Borland C++ entwickeln, können Sie THELP laden und die Online-Hilfe für ObjectWindows bei der Programmierung zu Rate ziehen.

Geben Sie zum Laden von THELP folgende Kommandozeile ein:

```
THELP /FC:\BORLANDC\OWL\OWLTHELP.OWH
```

Mit diesem Befehl wird THELP geladen und die Datei OWLTHELP.OWH als die zu verwendende Turbo-Hilfedatei festgelegt. Nach dem Laden rufen Sie THELP durch Drücken der Taste 5 auf dem Nummernblock auf.

Hinweis: Sie können diese Voreinstellung ändern; Informationen dazu finden Sie in der Online-Datei THELP.DOC.

THELP blendet ein Pop-up-Fenster ein und versucht das Wort im Hilfe-Index zu finden, auf dem oder neben dem der Cursor steht. Kann das Wort gefunden werden, wird der Hilfebildschirm für diesen Eintrag angezeigt. Andernfalls erhalten Sie den Hilfe-Index, so daß Sie den gewünschten Eintrag auswählen können.

THELP verfügt über folgende Eigenschaften:

Taste	Aktion
Tab	Bewegt den Cursor auf das nächste Schlüsselwort.
Umschalt Tab	Bewegt den Cursor auf das vorhergehende Schlüsselwort.
Posl	Springt zum Beginn der Zeile.
Ende	Springt zum Ende der Zeile.
CR	Wählt den Hilfeeintrag für das markierte Schlüsselwort im aktuellen Hilfebildschirm aus.
Esc	Beendet THELP.
F1	Zeigt den Hilfe-Index an.
Alt F1	Zeigt die letzten 20 Bildschirme, die Sie sich angesehen haben, in umgekehrter Reihenfolge an.
Strg F1	Zeigt einen Hilfebildschirm mit den THELP-Tastenkürzeln an.
Strg P	Fügt den Beispieltext in die Anwendung ein.

Sie können viele der THELP-Optionen ändern, indem Sie andere Kommandozeilenparameter eingeben. Informationen hierzu finden Sie in der Online-Datei THELP.DOC.

Die Typographie in diesem Handbuch

Zur Verdeutlichung von Zusammenhängen verwenden wir in diesem Handbuch mehrere unterschiedliche Schriftarten:

`Schreibmaschine` steht für Syntaxdefinitionen, Eingaben des Benutzers, Programmlistings und Ausgaben, die von Beispielprogrammen erzeugt werden.

Fettschrift kennzeichnet reservierte Wörter von Borland C++ wie **char, switch, near, cdecl,** usw. Innerhalb von Programmlistings sind reservierte Wörter nicht speziell gekennzeichnet.

Kursiv wird für Funktionsnamen (wie *printf*) sowie für Namen von Variablen und Bezeichnern im laufenden Text benutzt. Innerhalb von Programmlistings sind Funktionsnamen nicht speziell gekennzeichnet.

[Esc] für die Tasten Ihrer Tastatur, wie beispielsweise in »Ein Druck auf [Esc] entfernt das Menü wieder vom Bildschirm«.

[UP], [LEFT] für die Pfeiltasten zur Bewegung des Cursors und andere Spezialtasten.

Namenskonventionen in ObjectWindows

Die ObjectWindows-Namen für Klassen und Strukturen beginnen mit einem *T* (wie *TWindow*). Für alle *T*-Klassen definiert ObjectWindows folgende verwandte Typen:

P*class*, einen Zeiger auf die Klasse *class* (z.B. ist *PTWindow* definiert als Zeiger auf *TWindow*).

R*class*, eine Referenz auf die Klasse *class* (z.B. ist *RTWindow* definiert als Referenz auf *TWindow*).

RP*class*, eine Referenz auf einen Zeiger auf die Klasse *class* (z.B. ist *RPTWindow* definiert als Referenz auf einen Zeiger *TWindow*).

PC*class*, ein Zeiger auf die const-Klasse *class* (z.B. ist *PCTWindow* definiert als ein Zeiger auf die const-Klasse *TWindow*).

RC*class*, eine Referenz auf die const-Klasse *class* (z.B. ist *RCTWindow* definiert als eine Referenz auf die const-Klasse *TWindow*).

Diese Typen dienen der Vereinfachung komplizierter Windows-Programmierprozeduren. Ohne diese vordefinierten Typen würden dynamische Linkbibliotheken beispielsweise folgende Deklarationen erfordern:

```
TWindow _FAR *
```

In diesem Beispiel wird ein Zeiger auf ein *TWindow* deklariert.
Hinweis: Obige Deklaration ist äquivalent zu *PTWindow*.

Ein weiteres Beispiel ist die Referenz auf einen Zeiger auf ein *TWindow*:

```
TWindow _FAR * _FAR &
```

Hinweis: Obige Deklaration ist äquivalent zu *RPTWindow*.

Die Verwendung von *P-*, *R-*, *RP-* und *RC*-Typen bedeutet, daß Sie Ihre Anwendung sehr leicht konvertieren können, wenn Sie sie später als DLL definieren wollen. In den Beispielprogrammen und der -anwendung sind diese Typen intensiv eingesetzt.

Fragen und Probleme – die Beratung der Borland GmbH

Bei Problemen mit Borland C++ bitten wir Sie, zuerst die Handbücher und die Datei README (auf der Borland C++ Diskette 1) sorgfältig durchzulesen. Wenn danach Fragen offenbleiben, können Sie sich an eine der folgenden Stellen wenden:

1. Ihren Softwarehändler oder einen Benutzerclub:

2. In schriftlicher Form an:

Borland GmbH
Technische Beratung
Gautinger Straße 10
8130 Starnberg

3. In sehr dringenden Fällen an die Hotline der Borland GmbH:

Tel.: 089/7258001
Montag bis Donnerstag von 14 bis 17 Uhr
Freitag von 14 bis 16 Uhr

Sie sollten in allen Fällen die Handbücher sowie die folgenden Informationen parat haben:

1. Die Versionsnummer von Borland C++. (Die Kommandozeilenversion von Borland C++ gibt diese Nummer beim Start aus, innerhalb der integrierten Entwicklungsumgebung erzeugt die Eingabe von Alt F10 ein Fenster mit der entsprechenden Meldung).

2. Produktnamen und Modellbezeichnung Ihres Computers sowie Namen eventueller Zusatz-Hardware.

3. Den Namen des Betriebssystems (PC-DOS oder MS-DOS) sowie die entsprechende Versionsnummer. (Diese Nummer erhalten Sie durch die Eingabe des Befehls VER auf der DOS-Kommandoebene.)

I. Eine Einführung in ObjectWindows

1. Das Fenster vererben

Das vorliegende Kapitel enthält einen Überblick über die Progammierung für Windows, wobei der objektorientierten Programmierung besonderes Gewicht zufällt. Sie erfahren in diesem Kapitel, woraus eine Windows-Anwendung besteht, wie sie sich verhalten muß und wie durch objektorientiertes Programmieren mit ObjectWindows Arbeitsabläufe automatisiert und vereinfacht werden können.

Was ist eine Windows-Anwendung?

Abbildung 1.1 zeigt die Hauptkomponenten einer Windows-Anwendung. Die Kenntnis dieser Komponenten ist Voraussetzung für das Verständnis des vorliegenden Handbuchs.

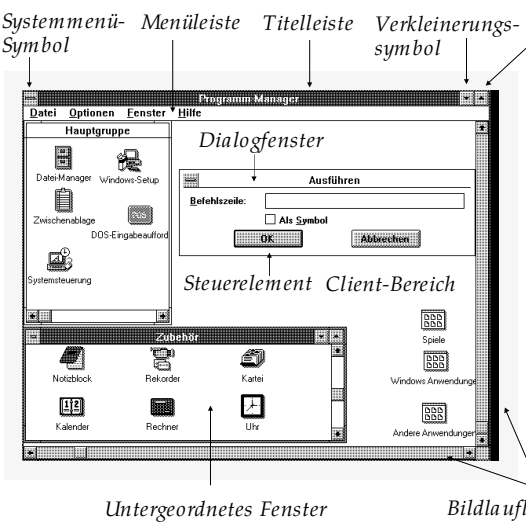


Abbildung 1.1 Die Bildschirmkomponenten einer Windows-An-

Bei einer Windows-Anwendung handelt es sich um einen speziellen Typ eines PC-Programms mit folgenden Eigenschaften:

- Es muß in einem ausführbaren Dateiformat vorliegen (spezielle .EXE-Datei).
- Es kann nur unter Windows ausgeführt werden.
- Es läuft im allgemeinen in einem rechteckigen Bildschirmfenster ab.
- Es reagiert auf Eingaben des Benutzers in einer standardisierten Art und Weise.
- Es kann gleichzeitig mit anderen Windows-Anwendungen oder sonstigen Programmen, einschließ lich weiterer Instanzen von sich selbst, ablaufen.
- Es kann mit anderen Windows-Anwendungen kommunizieren und Daten gemeinsam nutzen.

Vorteile von Windows

Windows bietet sowohl Anwendern als auch Programmierern viele Vorteile. Für den Anwender bringt es folgenden Nutzen:

- Standardisierte und konsistente Programmoberfläche. Wenn Sie eine Windows-Anwendung bedienen können, finden Sie sich in den meisten Windows-Anwendungen zurecht.
- Das Setup für Peripheriegeräte entfällt, da Windows Gerätetreiber zur Verfügung stellt.
- Kooperation und Kommunikation zwischen verschiedenen Anwendungen.
- Multitasking, d.h. die Möglichkeit zur gleichzeitigen Ausführung mehrerer Anwendungen.
- Größ ere Speicherkapazität: Windows unterstützt den *Protected Mode* bei 80286-, 80386- und i486- und virtuellen Speicher bei 80386- und i486-Computern.

Der Programmierer hat folgende Vorteile:

- Geräteunabhängige Grafik; Grafikanwendungen können mit allen Standard-Grafikkarten ausgeführt werden.
- Unterstützung einer Vielzahl von Druckern, Bildschirmen, Mäusen und anderen Peripheriegeräten.
- Eine umfangreiche Bibliothek mit Grafikroutinen.
- Mehr Programm- und Datenspeicher.
- Unterstützung von Menüs, Symbolen, Bitmaps etc.

Anforderungen an die Hardware

Den vielen Vorteilen von Windows stehen gewisse Anforderungen an die Hardware gegenüber. Windows setzt einen höheren Standard hinsichtlich der

Grafikfähigkeiten und der Speicherkapazität voraus und erfordert einen schnelleren Prozessor, um dieselbe Ausführungsgeschwindigkeit wie bei einer DOS-Anwendung zu erreichen. Mindestvoraussetzung für den effektiven Einsatz von Windows ist ein 80286-Computer mit wenigstens 2 MByte Arbeitsspeicher.

Objektorientierte Programmierung für Windows

Die Programmierung für eine fensterorientierte Umgebung basiert auf vielen neuen Konzepten. Zu Beginn kann die Erstellung eines »einfachen« Windows-Programms entmutigend sein. ObjectWindows vereinfacht diesen Prozeß, weil sich der Programmierer auf die Funktion des Programms konzentrieren kann und die Form erst in zweiter Linie berücksichtigen muß.

Mit ObjectWindows hat der Programmierer einen Rahmen zur Verfügung, in dem er mit Hilfe von Objekten die komplexen Schnittstellenelemente einer Windows-Anwendung darstellen kann. Die ObjectWindows-Fensterobjekte enthalten die Daten, die für alle Fenster erforderlich sind, führen allgemeine Fensteroperationen aus und antworten auf allgemeine Windows-Botschaften und -Ereignisse. Die ObjectWindows-Fenster- und -Anwendungsklassen übernehmen die Botschaftsabwicklung, die normalerweise den größten Teil einer Windows-Anwendung ausmacht.

Eine bessere Schnittstelle zu Windows

ObjectWindows setzt die objektorientierten Erweiterungen von Borland C++ für die Kapselung der Windows-Schnittstelle für die Anwendungsentwicklung (API) ein. Dies erleichtert die Arbeit des Programmierers wesentlich, da er sich nicht mit den Details der Windows-Programmierung befassen muß. ObjectWindows stellt drei hilfreiche Werkzeuge zur Verfügung:

- Die Kapselung von Fensterdaten.
- Abstraktion vieler Funktionen der Windows-API.
- Die Beantwortung von Botschaften.

Kapselung

ObjectWindows stellt Objekte zur Verfügung, die das Verhalten und die Eigenschaften von Fenstern, Dialogfenstern und Steuerelementen einer Windows-Anwendung definieren. In einer Windows-Anwendung dient ein *Schnittstellenobjekt* zur Repräsentation eines visuellen Windows-Schnittstellenelements. Schnittstellenobjekt und -element hängen eng zusammen, trotzdem bestehen Unterschiede zwischen beiden, die Sie kennen sollten.

Ein Schnittstellenelement wird mit einem neuen Schnittstellenobjekt durch den (indirekten) Aufruf seiner Elementfunktion *Create* angelegt. *Create* ruft die

zur Erstellung eines neuen Schnittstellenelements geeignete Windows-Funktion auf. Datenelemente des Schnittstellenobjekts, die in dem Windows-Aufruf übergeben werden, definieren die physikalischen Attribute des zu erstellenden Schnittstellenelements. Windows liefert das Handle oder den Bezeichner des erstellten Schnittstellenelements zurück. Das Schnittstellenobjekt speichert dieses Handle in seinem Datenelement *HWindow*. Dadurch, daß das Handle in einem Datenelement gespeichert ist, kann ohne größere Umstände darauf zugegriffen werden. Außerdem können diese Datenelemente zum Speichern von Zeichenwerkzeugen und Statusinformationen verwendet werden.

Die Abstraktion von Windows-Funktionen

Eine Windows-Anwendung bestimmt ihre Oberfläche und ihr Verhalten durch den Aufruf von Windows-Funktionen. Dafür stehen fast 600 Funktionen zur Verfügung, genannt Windows-API (Application Programming Interface). Jede Funktion benötigt eine Vielzahl von Parametern verschiedenster Art. Die Funktionen könnten zwar auch direkt von Borland C++ aufgerufen werden, ObjectWindows vereinfacht diese Aufgabe wesentlich, indem es Objekt-Elementfunktionen zur Abstraktion der Funktionsaufrufe zur Verfügung stellt.

Wie erwähnt, sind viele der Parameter für Windows-Funktionen bereits in den Datenelementen der Schnittstellenobjekte gespeichert. Die Elementfunktionen können diese Daten als Parameter für Windows-Funktionen verwenden. Zusätzlich gruppiert ObjectWindows verwandte Funktionsaufrufe zu einzelnen Elementfunktionen, die dann komplexere Aufgaben ausführen. Das Ergebnis ist eine vereinfachte Schnittstelle zu Windows.

Dieser Ansatz reduziert zwar die Abhängigkeit des Programmierers von der Vielzahl der Windows-API-Funktionen, blockiert aber nicht den direkten Zugriff auf die API. ObjectWindows bietet somit den Vorteil einer objektorientierten Programmierung *und* der maximalen Kontrolle über die grafische Umgebung.

Automatische Botschaftsbeantwortung

Die meisten Anwendungen müssen außer der Weitergabe von Anweisungen an Windows die Fähigkeit besitzen, auf die vielen Windows-Botschaften zu antworten, die sich aus Benutzereingaben, wie zum Beispiel einem Mausklick, anderen Programmen oder sonstigen Quellen ergeben. Bedenken Sie, daß Windows jedesmal eine Botschaft an Ihre Anwendung übergibt, wenn der Benutzer mit der Maus klickt oder sie auch nur bewegt. So ist es gar nicht selten, daß Ihre Anwendung während weniger Minuten tausende von Botschaften empfängt. Die korrekte Bearbeitung und Beantwortung derartiger Botschaften ist für die einwandfreie Funktion des Programms unerläßlich. Es ist zwar relativ einfach, ein Programm zu veranlassen, auf eine Menüauswahl in einer bestimmten Weise zu reagieren und auf eine Botschaft zu antworten. Ein Programm zu schreiben, das auf die annähernd 200 verschiedenen Windows-Botschaften antworten kann, ist aber genau so schwierig, wie die richtigen Windows-Funktionen aufzurufen.

Objekte, die Verhalten und Eigenschaften in Form von Elementfunktionen erben und überschreiben, sind optimal für die Beantwortung eingehender Stimuli (Windows-Botschaften) geeignet. ObjectWindows empfängt Windows-Botschaften und konvertiert sie in Elementfunktionsaufrufe von Borland C++. Mit Hilfe von ObjectWindows kann also eine Elementfunktion zur Beantwor-

tung jeder Botschaft definiert werden, die von der Anwendung empfangen wird. Drückt der Anwender beispielsweise die linke Maustaste, erzeugt Windows eine Botschaft namens *WM_LBUTTONDOWN*. Soll ein Fenster oder ein Steuerelement der Anwendung auf derartige Mausklicks reagieren, kann dazu eine *WMLButtonDown*-Elementfunktion definiert werden, die auf *WM_LBUTTONDOWN*-Botschaften antwortet. Das Objekt ruft dann die definierte Elementfunktion auf, wenn Windows diese Meldung übergibt.

Derartige Elementfunktionen werden als *Botschaftsantwortfunktionen* bezeichnet. Sie ordnen mit Hilfe einer dynamsichen virtuellen Tabelle einer Windows-Botschaft eine Funktion zu, die diese beantwortet. Die Tabelle hat den langen Namen *dynamic dispatch virtual table* und wird mit DDVT abgekürzt. Die Zuordnung erfolgt über einen *Index* vom Typ **Integer**, den Sie bei der Deklaration angeben. Im folgenden Beispiel

```
virtual void Test() = [100];
```

ist *Test* als Klassen-Elementfunktion mit einem Index von 100 definiert.

Wenn Sie sich windows.h ansehen, werden Sie feststellen, daß Windows-Botschaften über beschreibende Namen, wie *WM_LBUTTONUP* verfügen. Sie sind in Wirklichkeit aber Integerwerte, die Sie als Index für Botschaftsantwortfunktionen verwenden können.

Eine ObjectWindows-Anwendung schickt eine Windows-Botschaft an die Funktion weiter, deren Index dem Wert der Botschaft entspricht. Das folgende Beispiel

```
virtual void WMLButtonUp(RTMessage Msg)= [WM_FIRST + WM_LBUTTONUP];
```

deklariert eine Funktion vom Typ **void**, die auf die Windows-Botschaft *WM_FIRST + WM_LBUTTONUP* antwortet. Sie müssen nichts weiter tun – ObjectWindows weiß, wann Sie eine neue Botschaftsantwortfunktion hinzugefügt haben und ruft diese bei Auftreten der entsprechenden Botschaft auf.

Die Summe der Konstanten *WM_FIRST* und *WM_LBUTTONUP* ergibt den Integerwert, den Windows zur Identifizierung einer Botschaft verwendet.

ObjectWindows erwartet, daß Botschaftsantwortfunktionen einen Parameter vom Typ *RTMessage* haben. Zur Erinnerung: Eine *RTMessage* ist eine Referenz auf ein *TMessage*. *TMessage*-Objekte enthalten alle Informationen, die Windows in einer Botschaft übergibt.

Wichtig: Bei der Verwendung einer Botschaftsantwortfunktion gibt es einige Einschränkungen:

- Wenn eine Botschaftsantwortfunktion einer Basisklasse in einer abgeleiteten Klasse überschrieben wird, muß derselbe Index verwendet werde. Das folgende Beispiel ist aus diesem Grund falsch:

```
class TestBase
{
  virtual void sample() = [100];
};

class TestDerived : public TestBase
{
```

```
  virtual void sample() = [299];     /* Fehler */
};
```

- Jede Botschaftsantwortfunktion in einer Klassen-Hierarchie muß einen (umkehrbar) eindeutigen Index haben. Das folgende Beispiel ist nicht zulässig:

```
class TestBase
{
  virtual void sample1() = [100];
  virtual void sample2() = [200];
  virtual void sample3() = [100];
};
```

Diese Anweisung ist falsch, weil sie versucht, zwei verschiedenen Funktionen denselben Index zuzuordnen.

- Eine »normale« virtuelle Elementfunktion in einer Basisklasse kann in einer abgeleiteten Klasse nicht als Botschaftsantwortfunktion überschrieben werden. Folgendes Beispiel ist daher nicht zulässig:

```
class TestBase
{
  virtual void sample();
};

class TestDerived : public TestBase
{
  virtual void sample() = [100];   /* Fehler */
}
```

- Eine dynamische Botschaftsantwortfunktion in einer Basisklasse kann in einer abgeleiteten Klasse nicht als eine »normale« virtuelle Funktion überschrieben werden. Das folgende Beispiel ist deshalb nicht zulässig:

```
class TestBase
{
  virtual void sample() = [100];
};

class TestDerived : public TestBase   /* Fehler */
{
  virtual void sample();
};
```

- Wenn eine Klasse von mehr als einer einzigen Klasse abgeleitet ist, wird die abgeleitete Klasse als mehrfach vererbte Klasse bezeichnet. Nur die *erste* Basisklasse einer mehrfach vererbten abgeleiteten Klasse darf eine Botschaftsantwortfunktion enthalten. Folgende Definition ist aus diesem Grund zulässig:

```
class A
{
  virtual void AA() = [100];
};
```

```
class B
{
};

class C : puhblic A, public B
{
};
```

Folgende Definition ist nicht zulässig, weil die *zweite* Basisklasse von C, also die Klasse B, eine Botschaftsantwortfunktion hat.

```
class A
{
};

class B
{
  virtual void BB() = [100];
};

class C : public A, public B
{
};
```

- Wenn das Klassenobjekt über einen Zeiger angesprochen wird und kein spezieller Gültigkeitsbereich festgelegt ist, können Botschaftsantwortfunktionen nicht aufgerufen werden. Das folgende Beispiel ist nicht zulässig:

```
class Base
{
  virtual void sample() = [100];
}

int main(void)
{
  Base b;
  Base *BasePtr = &b;

  BasePtr->sample();
};
```

Diese Anweisung ist nicht zulässig, weil Sie eine dynamische Botschaftsantwortfunktion nicht direkt über einen Zeiger aufrufen können.

- Eine virtuelle Basisklasse darf keine dynamische Botschaftsantwortfunktion enthalten. Folgendes Beispiel ist daher nicht zulässig:

```
class Base
{
  virtual void sample() = [100];
};

class Derived : virtual public Base   /* Fehler */
{
};
```

Die alternative Methode zur Beantwortung von Windows-Botschaften, die ObjectWindows umgeht (aber in der Microsoft-C-Programmierung erforderlich ist), besteht in der Verwendung einer **switch**-Anweisung mit einem **case**-Zweig für *jede* Windows-Botschaft, auf die Ihr Programm antwortet. Das folgende Beispiel stammt aus einer in C geschriebenen Windows-Anwendung und enthält **switch**-Anweisungen innerhalb von **switch**-Anweisungen:

```
switch (message)
{
  case WMU_DDE_CLOSED:
    // Diese Botschaft wird von dem ddeclnt-Modul gesendet,
    // wenn der dynamische Datenaustausch beendet wird.
    CloseMsgWnd();
    break;

  case WM_COMMAND:
    switch (wParam)
    {
       case IDM_QUIT:
         // Der Anwender hat BEENDEN aus dem Menü gewählt.
         PostMessage(hWnd, WM_CLOSE, 0, 0L);
         break;
       case IDM_HOME:
         // Die Taste Pos1 wurde gedrückt.
         SendMessage(hWnd, WM_HSCROLL, SB_TOP, 0L);
                                 /* nicht aufgebaut */
         SendMessage(hWnd, WM_VSCROLL, SB_TOP, 0L);
         break;
       case IDM_ABOUT:
         // Der Anwender hat INFO aus dem Menü gewählt.
         lpproc = MakeProcInstance(About, hInst);
         DialogBox(hInst, MAKEINTRESOURCE(ABOUT), hWnd, lpproc);
         FreeProcInstance(lpproc);
         break;
         .
         .
         .
```

Bei bis zu 200 verschiedenen Windows-Botschaften ist diese Methode kaum mehr handhabbar. Die Verwendung der virtuellen Tabellen von ObjectWindows zur Beantwortung der Botschaften ist weitaus einfacher.

Die Struktur einer Windows-Anwendung

Bei der Vielzahl der Software-Elemente (DOS, Windows und andere Anwendungen), die miteinander in Wechselwirkung stehen, ist es hilfreich zu wissen, wie die Bestandteile einer Windows-Anwendung mit ihrer »Umgebung« zusam-

menarbeiten. Dieser Abschnitt befaß t sich mit der Struktur von Windows und von typischen Windows-Anwendungen, die mit Borland C++ unter Einsatz von ObjectWindows entwickelt wurden.

Die Struktur von Windows

Während der Ausführung befinden sich die Funktionen von Windows und der zugehörigen API in drei externen Bibliotheksmodulen, auf die bei Bedarf zugegriffen wird:

KERNEL.EXE ist verantwortlich für die Verwaltung des Speichers und der Ressourcen, für den Ablauf und für die Interaktion mit DOS.

GDI.EXE steuert die Ausgabe von Grafiken auf Bildschirm und Drucker.

USER.EXE übernimmt die Verwaltung der Fenster, der Anwendereingaben und die Kommunikation mit dem Anwender.

Diese Module sind Bestandteil von Windows, befinden sich also bereits auf den Rechnern der Anwender. Das Programm, das diese Module nutzt, müssen Sie als Programmierer zur Verfügung stellen.

Interaktion mit Windows und DOS

Aufgrund der Grenzen, die dem Betriebssystem DOS gesetzt sind, ist es relativ einfach, den Anteil von DOS an der erfolgreichen Ausführung von DOS-Programmen einzuschätzen.

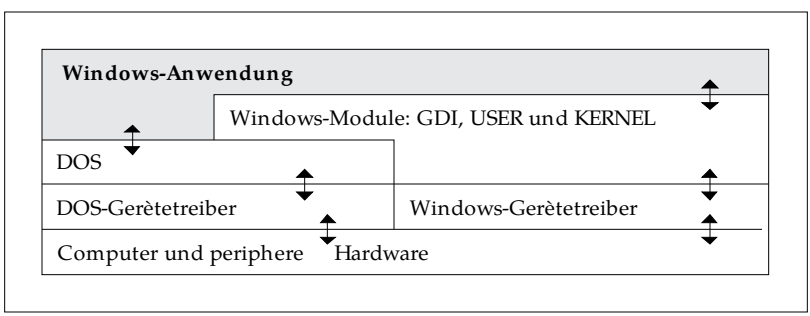


Abbildung 1.2 Interaktion von Windows-Anwendungen mit Windows und DOS. Der schattierte Bereich stellt das von Ihnen geschriebene Programm dar.

Ein DOS-Programm läuft aufgrund der Interaktion zwischen dem Programm und den Funktionen des Betriebssystems. Dies gilt auch für Windows-Anwendungen. Da aber Windows über weit mehr Betriebssystem-Funktionen verfügt als DOS, ist die Zusammenarbeit zwischen Windows und einer Anwendung

schwieriger zu überblicken. Das Programm muß also kontinuierlich mit dem Betriebssystem (DOS *und* Windows) zusammenarbeiten.

»Hello, Windows«

Eine neue Sprache oder Umgebung wird traditionellerweise mit dem Programm »Hello, World« eingeführt, das in der betreffenden Sprache bzw. für die entsprechende Umgebung geschrieben ist. Dieses Programm beschränkt sich normalerweise auf den Programmtext, der für die Anzeige des Strings »Hello, World« auf dem Bildschirm erforderlich ist.

Für Windows reicht dies nicht aus. Zunächst muß ein Fenster eingerichtet und in dieses Fenster geschrieben werden. Anschließend wird dem Fenster mitgeteilt, wie es mit seiner Umgebung agieren bzw. auf sie reagieren soll. Zumindest muß das Fenster wieder geschlossen und ausgeblendet werden können. Bereits für diese grundlegenden Schritte ist ein umfangreiches Programm erforderlich.

Der Grund für diesen Umfang liegt in den Anforderungen, die Windows selbst an das einfachste Programm stellt. Glücklicherweise erfüllen mit ObjectWindows geschriebene Programme die meisten dieser Anforderungen automatisch, unter anderem übernimmt ObjectWindows die Erzeugung und Anzeige des Hauptfensters. Das Programm »Hello World« läßt sich folgendermaßen verkürzen:

```
#include <owl.h>

// Define a class derived from TApplication
class THelloApp :public TApplication
{
public:
  THelloApp(LPSTR AName, HANDLE hInstance, HANDLE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
    : TApplication(AName, hInstance, hPrevInstance, lpCmdLine,
      nCmdShow){};
  virtual void InitMainWindow();
};

// Construct the THelloApp's MainWindow data member
void THelloApp::InitMainWindow()
{
  MainWindow = new TWindow(NULL, "Hello World!");
}
int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow)
{
  THelloApp HelloApp ("HelloApp", hInstance, hPrevInstance,
                      lpCmdLine, nCmdShow);
```

```
  HelloApp.Run();
  return HelloApp.Status;
}
```

Der Start einer Anwendung

Der erste Schritt nach dem Start eines ObjectWindows-Programms besteht in der Übergabe von vier Parametern an die Anwendung. In einem ObjectWindows-Programm werden diese Parameter an den Konstruktor Ihres von *TAppclication* abgeleiteten Anwendungsobjekts übergeben. Diese Parameter sind auch in Datenelementen des Anwendungsobjekts gespeichert, so daß Sie in Ihrem Quelltext problemlos darauf zugreifen können. Diese Datenelemente der Anwendung werden nachfolgend beschrieben:

hInstance speichert ein Handle auf die Anwendungsinstanz.

hPrevInstance speichert ein Handle auf die vorhergehende Instanz derselben Anwendung. Es ist 0, wenn es sich um die erste Instanz handelt.

lpCmdLine speichert einen *LPSTR*-Zeiger auf einen String mit dem Kommandozeileninhalt zum Start der Anwendung (einschließlich Optionen und Dateinamen). *lpCmdLine* könnte zum Beispiel die Zeilen "CALC.EXE /M" oder "WORDPROC.EXE LETTER1.DOC" enthalten.

nCmdShow speichert einen Integerwert, der den Anzeigemodus für das initiale Hauptfenster repräsentiert. Dieser Wert wird für Aufrufe der *Show*-Elementfunktion benötigt.

Als erstes muß eine ObjectWindows-Anwendung ein Anwendungsobjekt aus der Basisklasse *TApplication* erstellen. In unserem Beispiel verwenden wir dazu *THelloApp*.

Run ruft *InitApplication* nur für die erste Instanz einer ObjectWindowsAnwendung auf, um die erforderlichen Initialisierungsschritte für die gesamte Anwendung auszuführen. *Run* initialisiert alle Instanzen (einschließlich der ersten) durch den Aufruf von *InitInstance*. *InitInstance* ruft zur Initialisierung und Erstellung des Hauptfensterobjekts die Elementfunktion *InitMainWindow* auf.

Schließlich startet *Run* die Hauptbotschaftschleife der ObjectWindowsAnwendung. *Run* endet erst dann, wenn das Programm beendet wird. Der Exit-Status der Anwendung wird im Datenelement *Status* gespeichert. Dieser sollte an Windows zurückgegeben werden, wie in der besprochenen *WinMain*-Funktion.

Das Hauptfenster

Als Hauptfenster einer Anwendung bezeichnet man das Fenster, das nach dem Start der Anwendung angezeigt wird. Es stellt dem Anwender üblicherweise eine Liste mit den verfügbaren Befehlen – also ein Menü – zur Verfügung. Im Verlauf der Arbeitssitzung verwaltet das Hauptfenster die Schnittstelle der Anwendung und dient in vielen Fällen als alleinige Arbeitsfläche des Programms. Bei komplexeren Anwendungen kann die Arbeitsfläche aus mehreren Fenstern bestehen. Schließt der Anwender das Hauptfenster, beginnt damit der Vorgang zum Schließen der Anwendung.

Vorgehensweise bei der Anwendungsentwicklung

Da jede Windows-Anwendung bestimmte Anforderungen erfüllen muß (beispielsweise die Initialisierung des Hauptfensters), nimmt man am besten eine existierende Anwendung, und paßt diese an die eigenen Bedürfnisse an. ObjectWindows verfügt über mehrere Beispielprogramme, von denen Sie das für Sie passende auswählen können. Danach gehen Sie folgendermaßen vor:

1. Erstellen Sie den neuen Programmtext.
2. Erstellen Sie die Ressourcen für Menüs, Dialoge, etc.
3. Compilieren Sie das Programm.
4. Testen Sie das Programm.

2. Schritt für Schritt durch Windows

Nach dieser kurzen Einführung in die ObjectWindows-Bibliothek können Sie mit der Erstellung einfacher ObjectWindows-Programme beginnen. Dieses und die folgenden Kapitel enthalten Übungen, die Sie mit der Programmierung eines grafischen, interaktiven Windows-Programms vertraut machen sollen. Hierzu gehören

- die Erstellung von Menüs,
- das Speichern und Laden von Dateien,
- die Ausgabe von Grafiken und Text und
- das Anlegen eines einfachen Hilfesystems.

Sie erfahren außerdem Näheres über die wichtigsten Konzepte, die bei der Entwicklung einer Windows-Anwendung zu beachten sind, zum Beispiel die Verarbeitung von Botschaften, die Beziehungen zwischen über- und untergeordneten Fenstern und das Neuzeichnen von Grafiken.

In den Übungen werden folgende Arbeitsschritte durchgeführt:

- Schritt 1: Erstellen einer einfachen Anwendung.
- Schritt 2: Definieren einer Hauptfenster-Klasse.
- Schritt 3: Zeichnen von Text in einem Fenster.
- Schritt 4: Zeichnen von Linien in einem Fenster.
- Schritt 5: Ändern der Linienstärke.
- Schritt 6: Neuzeichnen der Fensterinhalte.
- Schritt 7: Hinzufügen eines Menüs.
- Schritt 8: Hinzufügen von Pop-up- und Dialogfenstern.
- Schritt 9: Speichern der Zeichnung in einer Datei.
- Schritt 10: Zeigen eines Hilfefensters.

Wenn Sie die bei der Installation voreingestellten Verzeichnisse nicht geändert haben, finden Sie den Quelltext im Unterverzeichnis EXAMPLES\STEPS des Hauptverzeichnisses OWL. Die Dateien haben die Namen STEP1.CPP, STEP2.CPP etc., je nachdem, welcher Schritt (englisch: step) gerade ausgeführt wird.

Abbildung 2.1 zeigt, wie die Anwendung nach der Durchführung aller Arbeitsschritte aussieht.

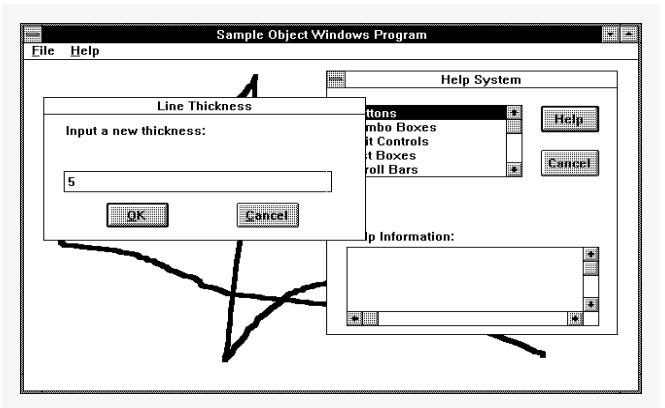


Abbildung 2.1 Eine ObjectWindows-Anwendung

Aufbau einer ObjectWindows-Anwendung: Vorbereitungen

Es gibt verschiedene Aspekte, die Sie beim Aufbau einer ObjectWindowsAnwendung beachten sollten. So müssen Sie als erstes die korrekten INCLUDE-Verzeichnisse angeben, damit der Compiler und der Ressourcencompiler die relevanten Dateien finden können. Als zweites müssen Sie die richtigen Bibliotheksverzeichnisse angeben, damit TLINK die für Ihre Anwendung erforderlichen Bibliotheken ausfindig machen kann. Schließ lich müssen Sie eine Ressourcendatei anlegen, die die geeigneten ObjectWindows-Ressourcen enthält und diese Ressourcendatei mit Ihrer Anwendung verbinden.

Die Container-Klassenbibliothek

Die Klassenbibliothek von ObjectWindows hängt von der Container-Klassenbibliothek im Unterverzeichnis CLASSLIB ab. Allen ObjectWindows-Objekten dient *Object* (definiert in der Container-Klassenbibliothek) als Basisobjekt. Andere Klassen in dieser Bibliothek werden ebenfalls von ObjectWindows-Klassen verwendet. Sie können alle Container-Klassen in Ihren ObjectWindowsAnwendungen einsetzen.

Verzeichnisse

Zusätzlich zu den für Ihre Anwendung erforderlichen C++-Standard-Headerdateien (wie stdio.h) muß den verschiedenen Compiler-Tools mitgeteilt werden, wo sich die ObjectWindows-Headerdateien, wie z.B. owl.h, befinden. Für das

Verfassen eines ObjectWindows-Programms muß zumindest die Datei owl.h. einbezogen werden. Komplexere Anwendungen erfordern zusätzliche Headerdateien. So muß in einem Programm, das die Klasse *TButton* verwenden soll, button.h vorhanden sein. ObjectWindows-Anwendungen und ObjectWindows selbst machen Gebrauch von der ObjectWindows-Container-Klassenbibliothek. Aus diesem Grund müssen Sie den Pfad des INCLUDE-Verzeichnisses für die ObjectWindows-Container-Klassenbibliothek angeben.

Wenn Sie ObjectWindows und Borland C++ an den im Installationsprogramm INSTALL vorgesehenen Positionen installiert haben, können Sie aus der folgende Tabelle die Standardverzeichnisse ersehen. Jede Anwendung muß auf die Standardbibliotheken von Borland C++ zugreifen. Außerdem muß der Zugriff auf selbsterstellte oder von anderer Seite bezogene Bibliotheken ermöglicht werden. ObjectWindows-Anwendungen benötigen zudem die Standardbibliotheken und die Container-Klassenbibliothek von ObjectWindows.

Wenn Sie das INSTALL-Programm von ObjectWindows oder Borland C++ angewiesen haben, andere Verzeichnisse anzulegen, müssen Sie die mitgelieferten Make- und Projektdateien Ihren dort gemachten Angaben entsprechend ändern.

Tabelle 2.1 Standard-INCLUDE-Verzeichnisse

Typ	Standard-Include-Verzeichnis	Standard-Bibliotheksverzeichnis
ObjectWindows	\BORLANDC\OWL\INCLUDE	\BORLANDC\OWL\LIB
Container-Klassenbibliothek	\BORLANDC\CLASSLIB\INCLUDE	\BORLANDC\CLASSLIB\LIB
Borland C++	\BORLANDC\INCLUDE	\BORLANDC\LIB

Wie bereits erwähnt, müssen Sie anstelle der mit Borland C++ ausgelieferten Container-Klassenbibliothek-Dateien die ObjectWindows-Container-Klassenbibliothek-Dateien verwenden.

Die korrekte Bibliothek festlegen

Es gibt drei verschiedene ObjectWindows- und Container-Klassenbibliotheken – jeweils eine Bibliothek für die Speichermodelle *Small*, *Medium* und *Large*. Diese Bibliotheken stehen in den oben erläuterten Bibliotheksverzeichnissen. Sehen Sie sich nun Tabelle 2.2 an.

Tabelle 2.2 Bibliotheken für die Speichermodelle

Speichermodell	ObjectWindows-Bibliothek	Container-Klassenbibliothek
Small	OWLWS.LIB	TCLASSWS.LIB
Medium	OWLWM.LIB	TCLASSWM.LIB
Large	OWLWL.LIB	TCLASSWL.LIB

ObjectWindows-Anwendungen mit dynamischen Linkbibliotheken

In der folgenden Tabelle finden Sie dynamische Linkbibliotheken (DLLs) und Importbibliotheken, die für ObjectWindows und die Borland-C++-Laufzeitbibliothek zur Verfügung stehen. Wenn Ihre Anwendung eine dieser dynamischen Linkbibliotheken verwenden soll, müssen Sie die entsprechende Importbibliothek dazu linken.

Tabelle 2.3 Dynamische Linkbibliotheken und Importbibliotheken

Typ	DLL	Importbibliothek
ObjectWindows	OWL.DLL	OWL.LIB
Borland C++	BCRTL.DLL	BCRTL.LIB

Sie können entweder beide oder keine, aber nicht nur eine dieser dynamischen Linkbibliotheken verwenden. Wenn Sie sie verwenden, müssen Sie Ihre Anwendung im Speichermodell *Large* (*-ml*) und mit der Windows-smart-callbacks-Option (*-WS*) compilieren. Benutzen Sie hierzu die angegebenen Optionen des Kommandozeilencompilers oder die entsprechenden Einstellungen unter *Options/Compiler* in der IDE. Da die Container-Klassenbibliothek in OWL.DLL enthalten ist, müssen Sie sie nicht explizit dazubinden.

Die Ressourcendatei erstellen

Normalerweise werden Sie einen Ressourceneditor zur Erstellung Ihrer Ressourcendateien verwenden. Sie können aber auch den Kommandozeilen-Ressourcencompiler RC dafür einsetzen. Eine ObjectWindows-Ressourcendatei ähnelt den anderen Windows-Ressourcendateien und kann ObjectWindows-spezifische Include-Dateien beinhalten. ObjectWindows verfügt über mehrere Ressourcendateien (mit den Dateinamenserweiterungen .DLG und .RC). Vier ObjectWindows-Klassen beziehen sich auf diese Ressourcendateien, die die in diesen Klassen verwendeten Ressourcen unterstützen. Sehen Sie sich dazu die folgende Tabelle an.

Tabelle 2.4 ObjectWindows-Klassen, die Ressourcendateien benötigen

Klassen	Erforderliche Ressourcendatei(en)
TInputDialog	INPUTDIA.DLG
TFileDialog	FILEDIAL.DLG
TEditWindow	STDWNDS.DLG, EDITACC.RC, EDITMENU.RC
TFileWindow	STDWNDS.DLG, FILEACC.RC, FILEMENU.RC

Wenn Ihre ObjectWindows-Anwendung eine der oben genannten Klassen verwendet, müssen Sie darauf achten, daß die Ressourcendatei für Ihre Anwendung die entsprechende ObjectWindows-Ressourcendatei einbindet. Beispiels-

weise könnte eine Ressourcendatei für eine Anwendung mit *TInputDialog* und *TFileDialog* folgendermaß en aussehen:

```
#include <windows.h>
#include <owlrc.h>
rcinclude inputdia.dlg
rcinclude filedial.dlg
...
```

Weitere für Ihre Anwendung erforderliche Ressourcen würden nach diesen Direktiven eingebunden werden. Die ObjectWindows-unterstützten Ressourcendateien und owlrc.h stehen im ObjectWindows-INCLUDE-Verzeichnis. Dieses Verzeichnis muß mit einer Kommandozeilenoption des Ressourcencompilers festgelegt werden. Da windows.h im Borland C++-INCLUDE-Verzeichnis steht, muß auch dieses Verzeichnis mit einer Kommandozeilenoption des Ressourcencompilers gesetzt werden. Verwenden Sie die Kommandozeilenoption *-i*, um INCLUDE-Verzeichnisse zu bestimmen. Sie können mehrere Verzeichnisse angeben, indem Sie die *-i*-Option wiederholen.

Aufbau einer ObjectWindows-Anwendung: die Einzelheiten

Jetzt sind alle Vorbereitungen für den Aufbau der Anwendung abgeschlossen. Sie können sich nun den speziellen Aufgaben für die Erstellung einer ObjectWindows-Anwendung zuwenden.

Sie können eine ObjectWindows-Anwendung auf drei verschiedene Arten aufbauen. Die erste (und einfachste) Möglichkeit besteht darin, das Ziel in einer der angegebenen Make-Dateien zu ändern. Die zweite Möglichkeit liegt in der Verwendung der integrierten Entwicklungsumgebung (IDE). Bei der dritten Möglichkeit setzen Sie die Kommandozeilen-Tools direkt ein. Wir befassen uns im folgenden mit den beiden letzteren Möglichkeiten. Der Einsatz einer Make-Datei ähnelt im Prinzip der Verwendung der direkten Kommandozeilen-Tools.

Wir gehen bei unseren Ausführungen davon aus, daß Sie mit den Borland-C++-Tools (IDE und Kommandozeilencompiler) vertraut sind. Sie sollten auch wissen, wie eine Windows-Anwendung im Prinzip aufgebaut wird. In der *Borland-C++-Dokumentation* finden Sie hierzu ausführliche Erläuterungen.

Die IDE verwenden

Zum Aufbau einer ObjectWindows-Anwendung mit Hilfe der Borland-C++-IDE müssen Sie eine *Projektdatei* verwenden. Detaillierte Erläuterungen zu Projekten finden Sie in der Dokumentation zu Borland C++. Im folgenden umreissen wir daher nur kurz die generellen Arbeitsschritte für den Aufbau eines ObjectWindows-Anwendungsprojekts:

- Sie können die Option für vorcompilierte Header-Dateien auswählen, um das Compilieren der vielen erforderlichen (und häufig umfangreichen) HeaderDateien zu beschleunigen.

- Öffnen Sie eine Projektdatei.

- Wählen Sie den Anwendungstyp *Windows EXE*.

- Wenn Sie die ObjectWindows-DLL oder eine andere dynamische Linkbibliothek wählen, müssen Sie den Modus »smart callbacks« und das Speichermodell *Large* wählen.

- Wählen Sie weitere geeignete Compiler- und Linkeroptionen aus.

- Bestimmen Sie die INCLUDE-Verzeichnisse (vgl. Sie dazu den Abschnitt »Verzeichnisse« weiter vorne in diesem Kapitel). Folgende Standard-INCLUDE-Verzeichnisse stehen zur Verfügung:

C:\BORLANDC\OWL\INCLUDE,

C:\BORLANDC\CLASSLIB\INCLUDE

C:\BORLANDC\INCLUDE

- Bestimmen Sie die Bibliotheksverzeichnisse. Folgende Verzeichnisse stehen Ihnen dafür standardmäß ig zur Verfügung:

C:\BORLANDC\OWL\LIB

C:\BORLANDC\CLASSLIB\LIB

C:\BORLANDC\LIB

- Bringen Sie die Quelltextdateien Ihrer Anwendung (mit den Namenserweiterungen .CPP, .C, .ASM, etc.) in das Projekt ein.
  Warnung: Sie dürfen keine Header-Dateien in das Projekt einfügen!

- Bringen Sie die .RC-Ressourcendatei in das Projekt ein. Weiter vorne in diesem Abschnitt finden Sie Informationen zu Ressourcendateien. In den folgenden Beispielen müssen Sie STEPS.RC in die Projektliste einbinden.
  Warnung: Sie dürfen keine .DLG-Dialogressourcendatei in das Projekt einfügen!

- Wenn Sie ObjectWindows statisch gelinkt in Ihrer Anwendung einsetzen wollen, müssen Sie die dem eingesetzten Speichermodell entsprechende Object-Windows-Bibliothek hinzufügen. So gehört beispielsweise OWLWS.LIB zum Speichermodell *Small*. Wenn Sie die ObjectWindows-DLL (OWL.LIB) verwenden möchten, fügen Sie die Importbibliothek OWL.LIB für ObjectWindows-DLLs hinzu.

- Fügen Sie die dem Speichermodell der Anwendung entsprechende Object-Windows-Container-Klassenbibliothek in das Projekt ein. So gehört beispielsweise TCLASSWM.LIB zu Anwendungen im Speichermodell *Medium*. Wenn Sie die ObjectWindows-DLL verwenden, brauchen Sie keine Container-Klassenbibliothek zu linken.

- Sie können die Borland-C++-Laufzeitbibliothek auch dynamisch zu Ihrer Anwendung linken, wodurch sich der Umfang der .EXE-Datei verringert. Fügen Sie zu diesem Zweck einfach die Importbibliothek CRTLL.LIB in Ihre Projektdatei ein.

- Wenn Sie statisch eine ObjectWindows-Bibliothek hinzulinken (OWLW*x*.LIB), müssen Sie auch die neue Bibliothek zum Reservieren von globalem Windows-Speicher statisch hinzulinken. Fügen Sie zu diesem Zweck einfach SALLOCW*x*.LIB (wobei *x* für das verwendete Speichermodell steht) in Ihre Projektdatei ein.

- Bauen Sie Ihre Anwendung auf. Die IDE-Projektverwaltung überprüft das Projekt daraufhin, ob die Quelltextdateien und die .RC-Ressourcendatei der Anwendung compiliert werden müssen; danach werden die .OBJ-Dateien und die Bibliotheken gelinkt; zuletzt werden die Ressourcen an die .EXE- oder .DLL-Datei angefügt.

Den Kommandozeilencompiler verwenden

Zum Compilieren und Linken Ihrer ObjectWindows-Anwendung mit Hilfe des Borland-C++-Kommandozeilencompilers BCC oder BCCX editieren Sie am besten Ihre Konfigurationsdatei und legen darin die INCLUDE- und Bibliotheksverzeichnisse fest. Geben Sie in dieser Datei die entsprechenden Verzeichnisse mit den Direktiven *-I* und *-L* an.

Hinweis: Weiter vorne in diesem Abschnitt finden Sie Informationen zu den INCLUDE- und Bibliotheksverzeichnissen.

Zum Compilieren und Linken Ihrer Anwendung geben Sie die folgende (oder eine ähnliche) Kommandozeile ein:

```
BCCX -WE myprog.cpp owlws.lib sallocws.lib tclassws.lib
```

Obige Kommandozeile compiliert zuerst MYPROG.CPP und linkt diese dann mit OWLWS.LIB, SALLOCWS.LIB und TCLASSWS.LIB.

Wenn Sie die ObjectWindows-DLL, die Borland-C++-Laufzeitbibliothek oder eine anwenderdefinierte ObjectWindows-DLL in Ihrer Anwendung einsetzen, müssen Sie die Compilierung mit der Kommandozeilenoption *-WS* (»smart callbacks«) durchführen. Beachten Sie, daß Sie für die Verwendung von ObjectWindows-DLLs Ihre Anwendung im Speichermodell *Large* aufbauen müssen. Sehen Sie sich folgendes Beispiel dazu an:

```
BCCX -WS -ml myprog.cpp owl.lib crtll.lib
```

Obige Kommandozeile compiliert MYPROG.CPP und linkt die compilierte Datei dann mit den Importbibliotheken OWL.LIB und CRTLL.LIB.

Danach müssen Sie die Ressourcendateien mit folgender (oder einer ähnlichen) Kommandozeile compilieren:

```
RC -r -i c:\borlandc\owl\include -i c:\borlandc\include myprog.rc
myprog.exe
```

Beachten Sie, daß die INCLUDE-Verzeichnisse in der RC-Kommandozeile explizit angegeben werden müssen. Sie könnten auch eine *Environment*-Variable mit dem Namen INCLUDE anlegen, um für *RC* die INCLUDE-Verzeichnisse festzulegen. So könnte das vorherige Kommando durch folgende Zeilen ersetzt werden:

```
SET INCLUDE=c:\owl\include;c:\borlandc\include

RC myprog.rc myprog.exe
```

Sie können die SET-Anweisung auch in die AUTOEXEC.BAT-Datei schreiben.

In allen Fällen compiliert RC die .RC-Ressourcendatei in eine binäre .RES-Ressourcendatei und fügt diese dann an die .EXE- oder .DLL-Datei Ihrer Anwendung an. Das Ergebnis ist eine ausführbare ObjectWindows-Anwendung.

Schritt 1: Eine einfache Windows-Anwendung

Sie beginnen Ihre Arbeit mit dem Schreiben einer äußerst einfachen ObjectWindows-Anwendung, die »Steps«-Anwendung genannt wird. Dieser Name bezieht sich auf die schrittweise (»step by step«) Weiterentwicklung des Programms.

Dieses Programm, es ist in der Datei STEP1.CPP enthalten, können Sie als Basis für alle Programme verwenden, die Sie mit ObjectWindows schreiben.

In Arbeitsschritt 1 wird die Steps-Anwendung das Hauptfenster der Anwendung initialisieren und erzeugen.

Voraussetzungen

Alle Windows-Programme verfügen über ein Hauptfenster, das angezeigt wird, wenn der Anwender das Programm startet. Das Verlassen des Programms erfolgt durch das Schließen des Hauptfensters. In einer ObjectWindows-Anwendung ist das Hauptfenster ein *Fensterobjekt*, das dem Anwendungsobjekt zugehörig ist. Das Anwendungsobjekt ist für die Erzeugung und das Anzeigen des Hauptfensters, die Bearbeitung von Windows-Botschaften und die Beendigung der Anwendung verantwortlich. Es agiert sozusagen als objektorientierter Ersatz für die Anwendung selbst. ObjectWindows stellt in gleicher Weise Objekttypen für Fenster, Dialogfenster und andere Klassen zur Verfügung, die Ihnen die Details der Windows-Programmierung ersparen.

Jedes ObjectWindows-Programm muß eine neue Anwendungsklasse definieren, die von der grundlegenden Klasse, *TApplication*, abgeleitet ist. In *WinMain* (dem Einsprungspunkt eines Windows-Programms) wird eine Instanz dieser abgeleiteten Klasse (das *Anwendungsobjekt*) erzeugt. Dabei ist festgelegt, daß Objekte normalerweise mit dem Buchstaben *T* und Zeiger auf Objekte mit dem Buchstaben *P* beginnen. In der Steps-Anwendung heißt diese Klasse *TMyApp*. Die Hauptfunktion der Steps-Anwendung sieht folgendermaßen aus:

```
int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
  LPSTR lpCmdLine, int nCmdShow)
{
  TMyApp MyApp("Sample ObjectWindows Programm", hInstance,
               hPrevInstance, lpCmdLine, nCmdShow);

  MyApp.Run();
  return MyApp.Status;
}
```

Hinweis: Dieses Programmfragment finden Sie in STEP1.CPP.

In der ersten Anweisung erstellt *WinMain* das *Anwendungsobjekt* der StepsAnwendung. *"Sample ObjectWindows Programm"* wird an den Konstruktor übergeben und dadurch zum Wert des Datenelements *Name*. Die anderen an *WinMain* übergebenen Parameter werden ebenfalls dem Konstruktor zur Verfügung gestellt und als Datenelemente im Anwendungsobjekt gespeichert. Danach wird mit dem Aufruf von *MyApp.Run* die ObjectWindows-Anwendung gestartet. Der Ende-Status (das *Status*-Datenelement des Anwendungsobjekts) wird in der letzten Anweisung zurückgeliefert.

Die Anwendungsklasse definieren

Ihre Anwendung muß eine neue Klasse von der Standard-ObjectWindows-Klasse *TApplication* (oder von einer von *TApplication* abgeleiteten Klasse) ableiten. Diese neue Klasse muß mindestens eine virtuelle Elementfunktion – *InitMainWindow* – überschreiben. *InitMainWindow* erstellt während der Initialisierung der ObjectWindows-Anwendung ein Hauptfensterobjekt.

Die Definition der Klasse *TMyApp* sieht folgendermaßen aus:

```
class TMyApp : public TApplication
{
public:
  TMyApp(LPSTR AName, HANDLE hInstance, HANDLE hPrevInstance,
         LPSTR lpCmdLine, int nCmdShow)
    : TApplication (AName, hInstance, hPrevInstance,
                    lpCmdLine, nCmdShow) {};
  virtual void InitMainWindow();
};
```

Hinweis: Dieses Prongrammfragment finden Sie in STEP1.CPP.

Das Hauptfensterobjekt wird im *MainWindow*-Datenelement des Anwendungsobjekts gespeichert. Zwar gehört das Hauptfensterobjekt zum Anwendungsobjekt, die beiden Objekte sind aber nicht hierarchisch miteinander verknüpft. Das Verhältnis der beiden Objekte zueinander wird als Instanzenbindung bezeichnet. Im folgenden sehen Sie die Definition von *TMyApp::InitMainWindow*:

```
void TMyApp::InitMainWindow()
{
   MainWindow = new TWindow(NULL, Name);
}
```

Hinweis: Dieses Programmfragment finden Sie in STEP1.CPP.

Das Hauptfensterobjekt von *TMyApp* wird im obigen Beispiel als Instanz der ObjectWindows-*TWindow*-Klasse erstellt. Der Parameter *NULL* bestimmt, daß es sich bei dem Fenster um das Hauptfenster der Anwendung handeln soll. Der zweite Parameter ist der Name oder Titel des Fensters. Das Datenelement *Name* des Anwendungsobjekts, das den String »Sample ObjectWindows Program« bereits enthält, wird an das neue *TWindow*-Objekt übergeben.

Ihr Hauptfensterobjekt wird normalerweise als Instanz einer Klasse erstellt, die von ObjectWindows-Klassen abgeleitet wurde. In Schritt 2 ersetzen Sie das hier erstellte generische Fenster durch ein passendes Fenster einer abgeleiteten Klasse. Abbildung 2.2 zeigt, wie die Steps-Anwendung auf dem Bildschirm dargestellt wird.

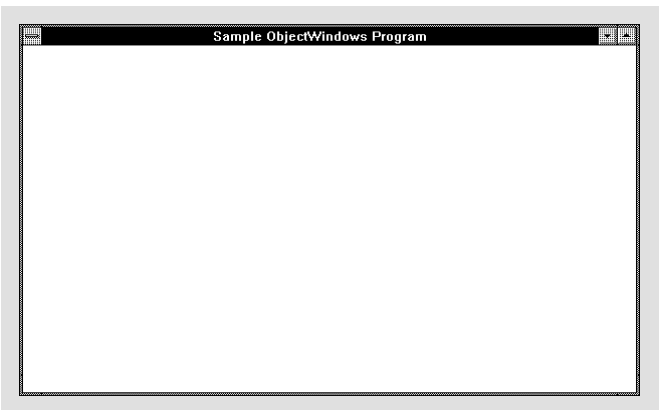


Abbildung 2.2
Ihr erstes ObjectWindows-Programm: Die StepsAnwendung

Bisher erzeugt die Anwendung nur ein leeres Fenster, das verschoben, vergrößert und verkleinert und geschlossen werden kann. Das vollständige (in STEP1.CPP enthaltene) Listing sieht momentan wie folgt aus:

```
#include <owl.h>

class TMyApp : public TApplication
{
public:
  TMyApp(LPSTR AName, HANDLE hInstance, HANDLE hPrevInstance,
         LPSTR lpCmdLine, int nCmdShow)
    : TApplication(AName, hInstance, hPrevInstance, lpCmdLine,
                   nCmdShow) {};
  virtual void InitMainWindow();
};

void TMyApp::InitMainWindow()
{
  MainWindow = new TWindow(NULL, Name);
}

int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow)
{
```

```
  TMyApp MyApp("Sample ObjectWindows Program", hInstance,
               hPrevInstance, lpCmdLine, nCmdShow);

  MyApp.Run();
  return MyApp.Status;
}
```

Schritt 2: Die Klasse für das Hauptfenster

Das mit Schritt 1 erstellte Programm besteht aus zwei Objekten: einem Anwendungsobjekt und einem Fensterobjekt. Das Anwendungsobjekt *MyApp* ist eine Instanz der Klasse *TMyApp*, die von *TApplication* abgeleitet ist. Der Fensterobjektzeiger, der im *MainWindow*-Feld des Anwendungsobjekts gespeichert ist, ist eine Instanz von *TWindow*, einer generischen ObjectWindows-Klasse. In fast allen (außer in ganz einfachen) Programmen müssen Sie die Fensterklasse für das Hauptfenster selbst definieren, und dabei das anwendungsspezifische Verhalten einarbeiten. In diesem Schritt werden wir dem Hauptfenster »Leben einhauchen«, indem wir eine komplexere, von *TWindow* abgeleitete Fensterklasse definieren.

Was ist ein Fensterobjekt?

In Schritt 1 haben wir ein Anwendungsobjekt kennengelernt, das alle Standardverhaltensweisen einer Windows-Anwendung zur Verfügung stellt, einschließlich der Erstellung des Hauptfensters. Die Klasse *TApplication* enthält alle grundlegenden Verhaltensweisen und Eigenschaften, die Sie für Ihre Windows-Anwendungen benötigen.

In ähnlicher Weise stellt ein Fensterobjekt die Verhaltensweisen der Fenster zur Verfügung, die von ObjectWindows-Anwendungen erzeugt werden. Zu diesen Eigenschaften gehören:

- Anzeige und Größenänderung;
- Antworten auf Ereignisse;
- Verwalten von untergeordneten Fenstern;
- ordnungsgemäßes Schließen.

Die ObjectWindows-Fensterklassen verfügen über diese Verhaltensweisen. Damit Ihre Programme effektiv und nützlich sind, müssen Sie neue, von *TWindow* abgeleitete Fensterklassen erzeugen. Diese neuen Klassen erben die Elementfunktionen und Datenelemente von *TWindow* und fügen neue hinzu. Der objektorientierte Ansatz erspart Ihnen also die ständig neue »Erfindung des Fensters«.

Handles

Alle Fensterobjekte haben mindestens drei Datenelemente: *HWindow*, *Parent* und die Liste der untergeordneten Fenster. *HWindow* enthält das Handle für das Fenster. Ein Handle verknüpft ein Schnittstellenobjekt (zum Beispiel ein Fenster, ein Dialogfenster oder ein Steuerelement) mit seinem entsprechenden Schnittstellenelement. Man könnte das Handle mit der Nummernmarke vergleichen, die Sie bei der Abgabe eines Kleidungsstücks an der Garderobe erhalten. So, wie Sie durch Vorweisen dieser Marke Ihr Kleidungsstück bezeichnen, bezeichnen Sie mit dem Handle das zugehörige Fenster.

Bei der Arbeit mit Fensterobjekten ist eine direkte Bezugnahme auf das Fenster-Handle normalerweise nicht erforderlich. Es wird aber benötigt, um eine Windows-Funktion direkt aufzurufen. Um zum Beispiel ein Meldungsfenster anzuzeigen, brauchen Sie nur die Funktion *MessageBox* aufzurufen. *MessageBox* erfordert die Angabe des Handle für das übergeordnete Fenster des Meldungsfensters. Ein *MessageBox*-Aufruf innerhalb einer Elementfunktion einer abgeleiteten Fensterklasse könnte folgendermaßen aussehen:

```
MessageBox(HWindow, "Do you want to save?",
           "Drawing has changed", MB_YESNO | MB_ICONQUESTION);
```

Eltern und Kinder

Die meisten Fenster einer Anwendung sind mit anderen Fenstern verknüpft. Wird zum Beispiel eine Anwendung beendet, muß sie alle Fenster entfernen, die sie angelegt hat. Windows teilt die Fenster in übergeordnete (Eltern) und untergeordnete Fenster (Kinder) ein. Ein übergeordnetes Fenster ist für seine untergeordneten Fenster verantwortlich. ObjectWindows stellt für jedes Fensterobjekt Datenelemente zur Verfügung, um die »Überwachung« des übergeordneten und der untergeordneten Fenster zu ermöglichen.

Die Beziehungen von übergeordneten Fenstern sind in Kapitel 5 »Einen Dialog erstellen« in Schritt 8 erläutert.

Das *Parent*-Datenelement enthält das Objekt des übergeordneten Fensters. Es handelt sich dabei weniger um ein Verwandtschafts- als um ein Zugehörigkeitsverhältnis. Diese Beziehungen zwischen den Fenstern werden in Schritt 8 beschrieben.

Die Liste der untergeordneten Fenster, das dritte Datenelement des Fensterobjekts, enthält eine verkettete Liste mit Zeigern auf die untergeordneten Fenster (falls derartige Fenster vorhanden sind). Wir werden in Schritt 8 untergeordnete Fenster hinzufügen.

Eine neue Fensterklasse erstellen

Wir wissen nun, was in einem Fensterobjekt enthalten ist, und können eine neue Fensterklasse entwerfen. Sie stammt von der Klasse *TWindow* ab und dient als Hauptfenster für die Steps-Anwendung.

Dazu aktualisieren wir zuerst die Klassendeklarationen für die neue Klasse *TMyWindow*.

```
_CLASSDEF(TMyWindow)

class TMyWindow : public TWindow
{
public :
  TMyWindow(PTWindowsObject AParent, LPSTR ATitle)
           : TWindow(AParent, ATitle){};
};
```

Hinweis: Dieses Programmfragment finden Sie in STEP2.CPP.

Beachten Sie, daß der Konstruktor für unsere neu abgeleitete Klasse keine neuen Verhaltensweisen definiert und lediglich den Konstruktor des Basisobjekts (*TWindow*) aufruft.

Anschließend aktualisieren wir *TMyApp::InitMainWindow*, damit ein *TMyWindow* und nicht ein *TWindow* als Hauptfenster erzeugt wird.

```
void TMyApp::InitMainWindow()
{
   MainWindow = new TMyWindow(NULL, Name);
}
```

Hinweis: Dieses Programmfragment finden Sie in STEP2.CPP.

Die Deklaration der neuen Klasse und ihre Instantiierung in *InitMainWindow* reicht aus, um eine neue Klasse für das Hauptfenster von *TMyApp* zu definieren. Das Anwendungsobjekt ruft Elementfunktionen des Fensterobjekts zur Erzeugung des Schnittstellenelements für das Fenster und zu dessen Anzeige auf dem Bildschirm auf.

Beachten Sie, daß *TMyWindow* keine neuen Verhaltensweisen definiert, sondern nur über diejenigen von *TWindow* und *TWindowsObject* verfügt. Mit anderen Worten, unsere Anwendung wird dadurch nicht interessanter.

Botschaften beantworten

Einem Fensterobjekt läßt sich am einfachsten eine sinnvolle Aufgabe zuweisen, indem man festlegt, wie es auf Windows-Botschaften zu antworten hat. Klickt der Anwender beispielsweise mit der linken Maustaste im Hauptfenster der Steps-Anwendung, erhält das entsprechende Fensterobjekt von Windows eine *WM_LBUTTONDOWN*-Botschaft. Diese teilt dem Fensterobjekt mit, daß sein Bereich vom Anwender mit der Maus angeklickt wurde. Außerdem übergibt sie die Koordinaten des Punkts, an dem der Mausklick stattfand.

Diese Informationen werden in Schritt 6 verwendet. Ähnliches liegt vor, wenn der Anwender mit der rechten Maustaste klickt. Das Hauptfensterobjekt erhält dann von Windows die Botschaft *WM_RBUTTONDOWN*. Unsere Aufgabe ist es nun, dem Hauptfenster – einer Instanz von *TMyWindow* – mitzuteilen, wie es auf diese Botschaften reagieren soll.

Windows-Botschaften werden abgefangen und beantwortet, indem man für jeden Typ eingehender (relevanter) Botschaften eine Elementfunktion definiert. Diese Elementfunktionen werden als *Botschaftsantwortfunktionen* bezeichnet. Sie

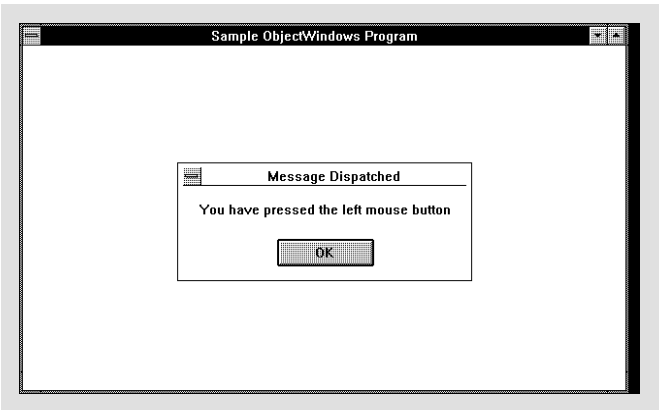


Abbildung 2.3
Die Antwort der StepsAnwendung auf eine Benutzereingabe

werden für jede Windows-Botschaft, die beantwortet werden soll, eine Botschaftsantwortfunktion definieren. Wenn Sie für eine bestimmte Botschaft keine Antwort definieren, führt ObjectWindows seine Standardbearbeitung aus. Um eine Elementfunktion als Botschaftsantwortfunktion zu kennzeichnen, fügen Sie in die Deklaration einen Index ein (vgl. in Kapitel 1 den Abschnitt »Automatische Botschaftsbeantwortung«).

Beachten Sie, daß der Name der Funktion im Gegensatz zum Wert des Index keine Rolle spielt. Wir verwenden für die Namen von Botschaftsantwortfunktionen standardmäßig die groß geschriebenen Anfangsbuchstaben der Windows-Botschaft, lassen aber den Unterstrich weg.

```
class TMyWindow : public TWindow {
public :
.
.
.
virtual void WMLButtonDown(RTMessage Msg) =
  [WM_FIRST + WM_LBUTTONDOWN];
.
.
.
};
```

Hinweis: Dieses Programmfragment finden Sie in STEP2.CPP.

Msg ist eine Referenz auf eine *TMessage*-Struktur (abgekürzt als ObjectWindows-Typ *RTMessage*), die Informationen über das aktuelle Ereignis enthält. Wir befassen uns in Schritt 3 ausführlicher mit dem Parameter *Msg*. Allen Botschaftsantwortfunktionen wird *RTMessage* übergeben. Diese Funktionen sind vom Typ **void**.

Im Moment beschränken wir uns auf Elementfunktionen, die auf ein Klicken mit der Maus mit der Anzeige eines Meldungsfensters reagieren. Später werden wir weitere, zweckmäßigere Antworten hinzufügen.

Die Definition für die Antwortelementfunktion auf den Druck der linken Maustaste lautet wie folgt:

```
void TMyWindow::WMLButtonDown(RTMessage Msg)
{
  MessageBox(HWindow, "You have pressed the left mouse button",
             "Message Dispatched", MB_OK);
}
```

Hinweis: Dieses Programmfragment finden Sie in STEP2.CPP.

Den kompletten Quelltext für diesen Schritt finden Sie am Ende dieses Kapitels.

Ein Programm beenden

Unser Programm wird beendet, wenn der Anwender das Systemmenüsymbol im Hauptfenster – das kleine Quadrat in der oberen linken Ecke – doppelt anklickt. Das Fenster wird daraufhin sofort geschlossen und die Anwendung beendet. Diese Verhaltensweise eignet sich zwar gut für einfache Programme, ist aber nicht in allen Fällen erwünscht.

Haben Sie schon einmal eine Anwendung verlassen, ohne abzuspeichern? Ein gutes Programm sollte den Anwender beim Verlassen des Programms immer fragen, ob er seine Arbeit speichern will (falls dies noch nicht geschehen ist). Sie können Ihre ObjectWindows-Anwendungen ohne großen Aufwand mit dieser Funktion versehen. Wir wollen nun unsere Steps-Anweisung mit einer derartigen Sicherung versehen.

Versucht der Anwender, ein ObjectWindows-Programm zu verlassen, wird die virtuelle *CanClose*-Elementfunktion der Anwendungsklasse aufgerufen. *CanClose* ist eine *BOOL*-Funktion, die anzeigt, ob das Beenden des Programms zulässig ist (*TRUE*). Die von *TApplication* geerbte *CanClose*-Elementfunktion ruft standardmäßig die *CanClose*-Elementfunktion des Hauptfensterobjekts auf. In fast allen Fällen bestimmt das Hauptfensterobjekt, ob die Beendigung des Programms zulässig ist.

Unsere Hauptfensterklasse *TMyWindow* erbt von *TWindowsObject* eine *CanClose*-Elementfunktion, die wiederum für jedes seiner untergeordneten Fenster (falls vorhanden) eine *CanClose*-Elementfunktion aufruft. Gibt es keine untergeordneten Fenster (in unserem Programm ist das der Fall), liefert *CanClose* *TRUE* zurück.

Um das Schließverhalten einer Anwendung zu ändern, definieren wir eine neue *CanClose*-Elementfunktion für die Hauptfensterklasse:

```
BOOL TMywindow::CanClose()
{
  return MessageBox(HWindow, "Do you want to save?",
                    "Drawing has changed",
                    MB_YESNO | MB_ICONQUESTION) == IDNO;
}
```

Dieses Programmfragment stammt aus STEP2.CPP. Beim Versuch, die Steps-Anwendung zu beenden, erhält der Anwender jetzt ein Meldungsfenster mit der

Frage »Do you want to save?«. Anklicken des Yes-Schalters ergibt *FALSE* und verhindert, daß das Hauptfenster geschlossen und das Programm beendet wird. Anklicken des No-Schalters ergibt *TRUE* und beendet das Programm. Abbildung 2.4 zeigt die modifizierte Steps-Anwendung.

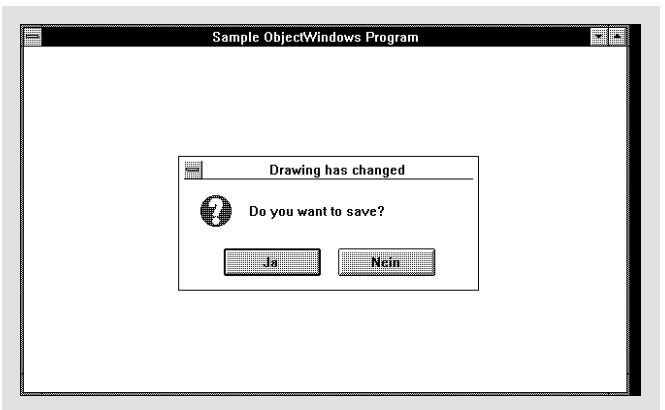


Abbildung 2.4
Die StepsAnwendung mit besserem Schließverhalten

Der vollständige (in STEP2.CPP enthaltene) Quelltext der Steps-Anwendung lautet bis jetzt:

```
#include <owl.h>

class TMyApp : public TApplication
{
public:
  TMyApp(LPSTR AName, HANDLE hInstance, HANDLE hPrevInstance,
         LPSTR lpCmdLine, int nCmdShow) : TApplication(AName,
         hInstance, hPrevInstance, lpCmdLine, nCmdShow) {};
  virtual void InitMainWindow();
};

_CLASSDEF(TMyWindow)
class TMyWindow : public TWindow
{
public:
  TMyWindow(PTWindowsObject AParent, LPSTR ATitle)
    : TWindow(AParent, ATitle) {};
  virtual BOOL CanClose();
  virtual void WMLButtonDown(RTMessage Msg)
    = [WM_FIRST + WM_LBUTTONDOWN];
  virtual void WMRButtonDown(RTMessage Msg)
    = [WM_FIRST + WM_RBUTTONDOWN];
};

BOOL TMyWindow::CanClose()
{
  return MessageBox(HWindow, "Do you want to save?",
```

```
    "Drawing has changed", MB_YESNO | MB_ICONQUESTION) == IDNO;
}

void TMyWindow::WMLButtonDown(RTMessage)

{
  MessageBox(HWindow, "You have pressed the left mouse button",
             "Message Dispatched", MB_OK);
}

void TMyWindow::WMRButtonDown(RTMessage)
{
  MessageBox(HWindow, "You have pressed the right mouse button",
             "Message Dispatched", MB_OK);
}

void TMyApp::InitMainWindow()
{
  MainWindow = new TMyWindow(NULL, Name);
}

int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow)
{
  TMyApp MyApp("Sample ObjectWindows Program", hInstance,
               hPrevInstance, lpCmdLine, nCmdShow);
  MyApp.Run();
  return MyApp.Status;
}
```

3. Ein Fenster füllen

Im folgenden wollen wir die Steps-Anwendung, die eigentlich erst der Rahmen für ein Programm ist, in eine brauchbare, interaktive Grafikanwendung verwandeln. Zuerst werden Sie Text in das Hauptfenster zeichnen. Anschließ end überführen wir die Steps-Anwendung in eine kleine Grafikanwendung, mit der der Anwender Linien im Hauptfenster zeichnen kann.

Schließ lich sorgen wir dafür, daß ein Neuzeichnen der Grafiken möglich ist, daß die Stärke von Linien geändert werden kann und daß Grafiken für eine spätere Wiederverwendung als Datei gespeichert werden können.

Was ist ein Bildschirmkontext?

Ein Bildschirmkontext repräsentiert die Zeichenoberfläche eines Fensters. Windows benötigt den Bildschirmkontext zum Zeichnen von Grafiken oder Text in einem Fenster. Ein Handle eines Bildschirmkontexts, sein Bezeichner, muß an die Funktionen der grafischen Geräteschnittstelle (GDI) übergeben werden. Zum Zeichnen in einem Fenster ist ein Bildschirmkontext erforderlich. Zu diesem Zweck rufen wir – bevor auf den Bildschirm gezeichnet wird – die Windows-Funktion *GetDC* in einer Elementfunktion des Fensterobjekts auf:

```
TheDC = GetDC(HWindow);
```

GetDC liefert das Bildschirmkontext-Handle (das vom Windows-Typ HDC ist), das Sie nun an die Windows-GDI-Funktionen übergeben können. Im folgenden sehen Sie einige Beispiele für GDI-Aufrufe:

```
LineTo(TheDC, 30, 45);

Rectangle(TheDC, 100, 100, 200, 200);

TextOut(TheDC, 20, 20, "Sample Text", 11);
```

Wenn das Zeichnen von Text oder Grafik beendet ist, *muß* der Bildschirmkontext freigegeben werden. Das unverzügliche Freigeben aller Bildschirmkontexte nach dem Zeichnen ist äuß erst wichtig:

```
ReleaseDC(HWindow, TheDC);
```

Wenn Sie die abgerufenen Bildschirmkontexte nach dem Zeichnen nicht freigeben, sind diese innerhalb kürzester Zeit aufgebraucht (die WindowsUmgebung verfügt nur über fünf Bildschirmkontexte). Dies hätte eine fehlerhafte Programmausführung (in den meisten Fällen einen Absturz) zur Folge. Stürzt Ihre Anwendung beim dritten oder vierten Zeichenvorgang ab, sollten Sie sie zuerst auf nicht-freigegebene Bildschirmkontexte überprüfen.

Der Bildschirmkontext dient zur Unterstützung wichtiger Zeichenfunktionen. Er stellt sicher, daß Text und Grafiken nicht auß erhalb der Fensteroberfläche gezeichnet werden. Text und Grafiken, die Sie durch einen Aufruf von GDI-Funktionen ausgeben, werden an den Grenzen des Fensters *abgeschnitten* (*clip*), auf das sich der Bildschirmkontext bezieht.

Der Bildschirmkontext verwaltet Zeichenwerkzeuge (Stift, Pinsel, Schrift, Palette), die beim Aufruf der GDI-Funktionen verwendet werden. In den vorausgehenden Beispielen wurde der im Bildschirmkontext ausgewählte Stift beim Aufruf von *LineTo* für das Zeichnen der Linie eingesetzt.

Die gezeichnete Linie hat die Eigenschaften des ausgewählten Stifts (einschließ lich Farbe und Stärke). Ebenso wird der im Bildschirmkontext ausgewählte Pinsel für das Ausfüllen eines festgelegten rechteckigen Bereichs verwendet, wenn *Rectangle* aufgerufen wird. Die ausgewählte Schrift wird beim Aufruf von *TextOut* für die Anzeige von "Sample Text" eingesetzt.

Sie können neue Werkzeuge für den Bildschirmkontext auswählen, um damit die Anzeige der Ausgabe zu ändern. So könnten Sie einen Pinsel mit einem Muster, einen Stift mit einer anderen Farbe oder eine andere Schrift auswählen. Die Auswahl eines neuen Stifts wird später in diesem Kapitel (in Schritt 5) erläutert.

Schritt 3: Text in einem Fenster zeichnen

Um Text in das Hauptfenster der Steps-Anwendung zu zeichnen, benötigen wir einen Bildschirmkontext. Dazu muß ein Bildschirmkontext angefordert, die Anzeige-Operationen ausgeführt und der Bildschirmkontext wieder freigegeben werden (vgl. den vorherigen Abschnitt).

Um das Ganze interessanter zu gestalten, zeichnen wir den Text als Reaktion auf den Druck der linken Maustaste (diese war in Schritt 2 die Anzeige eines Dialogfensters). Die Antwort soll aus dem Zeichnen von Text (und nicht aus der Anzeige eines Meldungsfensters) bestehen, der die Positionskoordinaten des Mausklicks angibt.

So wird z.B. »(20,30)« gezeichnet, wenn mit der Maus 20 Pixel rechts vom linken und 30 Pixel unter dem oberen Rand der Zeichenoberfläche des Fensters (*Client-Bereich*) geklickt wird. Der Text wird an dem Punkt gezeichnet, der mit der Maus angeklickt wurde.

Beachten Sie, daß das Klicken mit der linken Maustaste eine *WM_LBUTTONDOWN*-Botschaft erzeugt, die mit der Botschaftsantwortfunktion *WMLButtonDown* abgefangen wird.

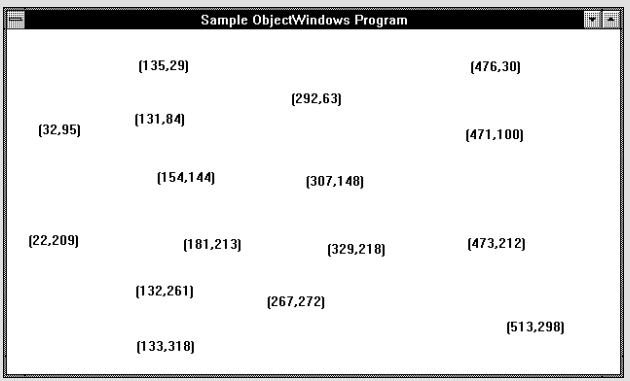


Abbildung 3.1 Die StepsAnwendung zeichnet Text an einem angeklickten Punkt

Botschafts-Struktur

In Schritt 2 haben wir gesehen, daß das*Msg*-Argument, eine *TMessage*-Struktur-Referenz (abgekürzt *RTMessage*) einer Botschaftsantwortfunktion, Informationen über das eingetretene Ereignis enthalten kann. Die Daten stehen in den Struktur-Elementen *LParam* und *WParam* und sind vom Typ *LONG* und *WORD*.

Msg.LParam enthält häufig zweierlei Informationen und zwar in seinem höher- und seinem niederwertigen *WORD*. *LParam* ist in einer Variante zusammen mit einer *LP*-Struktur deklariert, die *Hi* und *Lo* als Elemente vom Typ *WORD* definiert. Aus diesem Grund können Sie mit *Msg.LP.Hi* und *Msg.LP.Lo* auf die höher- und niederwertigen *WORD*-Elemente von *Msg.LParam* zugreifen. Das Struktur-Element *WParam* ist in einer Variante zusammen mit einer *WP*-Struktur deklariert. Die Elemente *Hi* und *Lo* der *WP*-Struktur werden jedoch nur selten verwendet.

Im Fall von *WMLButtonDown* enthält *Msg.LP.Lo* die x-Koordinate des angeklickten Punkts, während *Msg.LP.Hi* die y-Koordinate speichert. Wir wollen nun *WMLButtonDown* so schreiben, daß es einen Text mit den Koordinaten des angeklickten Punkts zeichnet. Dazu müssen *Msg.LP.Lo* und *Msg.LP.Hi* in Text konvertiert und mit Interpunktionszeichen versehen werden. In unserem Beispiel verwenden wir *sprintf* zum Aufbau des zu zeichnenden Textstrings.

Sobald der endgültige String aufgebaut ist, können Sie ihn an den angeklickten Punkt zeichnen, indem Sie – in einer *TextOut*-Funktion – den Bildschirmkontext, die Koordinaten, *Msg.LP.Lo* und *Msg.LP.Hi*, den String *S* und die Stringlänge, *strlen*(*S*) an Windows übergeben. Vor dem Zeichnen muß ein Bildschirmkontext angefordert und danach wieder freigegeben werden.

```
void TMyWindow::WMLButtonDown(RTMessage Msg)
{
   HDC DC;
   char S[10];

   sprintf(S,"(%d,%d)", Msg.LP.Lo, Msg.LP.Hi);
```

```
    DC = GetDC(HWindow);
    TextOut(DC, Msg.LP.Lo, Msg.LP.Hi, S, strlen(S));
    ReleaseDC(HWindow, DC);
}
```

Hinweis: Dieses Programmfragment finden Sie in STEP3.CPP.

Den Bildschirm löschen

Sie können dieser Anwendung zum Zeichnen von Text eine weitere Funktion zum Löschen des Bildschirms hinzufügen. Es ist möglich, gezeichneten Text durch das Vergrößern oder Verkleinern oder durch das Verbergen und Wiedersichtbarmachen des Fensters zu löschen. Es ist aber sinnvoller, dem Anwender das Löschen des Bildschirms über eine Menüauswahl oder eine andere Tätigkeit (zum Beispiel Klicken mit der Maus) zu ermöglichen.

In unserem Programm soll der Bildschirm durch das Drücken der rechten Maustaste gelöscht werden. Wir erreichen dies, indem wir die *WMRButtonDown*-Elementfunktion so überschreiben, daß sie die Windows-Funktion *InvalidateRect* aufruft, die ein Neuzeichnen des gesamten Fensters bewirkt. Da das Fenster noch nicht weiß, wie es sich selbst neu zeichnen muß, löscht es nur seinen Client-Bereich.

```
void TMyWindow::WMRButtonDown(RTMessage)
{
    InvalidateRect(HWindow, NULL, TRUE);
}
```

Der momentane Stand der Steps-Anwendung ist in der Datei STEP3.CPP enthalten.

Schritt 4: Linien in einem Fenster zeichnen

Nachdem Sie die Grundlagen für das Zeichnen auf dem Bildschirm kennen (Anfordern eines Bildschirmkontexts, Zeichnen und Freigabe des Bildschirmkontexts), können wir nun eine komplexere, interaktive Grafikanwendung erstellen. Die nächsten Schritte bestehen aus der Erstellung eines einfachen Programms, das dem Anwender das Zeichnen im Hauptfenster ermöglicht. Dazu wird folgendermaßen vorgegangen:

1. Antworten auf das Drücken der linken Maustaste und das Bewegen der Maus. Die Punkte werden entlang des Wegs verbunden, woraus sich gezeichnete Linien ergeben.

2. Antworten auf das Drücken der rechten Maustaste. Es wird ein Eingabefeld angezeigt, in dem der Anwender die Linienstärke ändern kann.

3. Automatisches Neuzeichnen des Fensterinhalts. Dazu werden die Punkte gespeichert und als Antwort auf eine *Paint*-Botschaft neu gezeichnet.

Bevor wir diese Schritte ausführen, wollen wir uns näher ansehen, wie in Windows die Bewegungen mit der Maus ausgeführt werden. Danach implementieren wir die einfache grafische Zeichenanwendung.

Bewegungen der Maus

Rufen Sie sich zunächst ins Gedächtnis, wie Windows auf Mausereignisse reagiert. Wie Sie bereits wissen, hat das Drücken der linken Maustaste eine *WM_LBUTTONDOWN*-Botschaft und den Aufruf der *WMLButtonDown* Elementfunktion zur Folge. Bisher haben wir auf das Drücken der linken Maustaste durch die Anzeige von Meldungsfenstern und das Zeichnen von Text geantwortet.

Ebenso ist bekannt, daß das Drücken der rechten Maustaste eine *WM_RBUTTONDOWN*-Botschaft und den Aufruf der *WMRButtonDown*-Elementfunktion zur Folge hat. Die Antwort darauf war das Löschen des Bildschirms. Diese Antworten betrafen alle nur den jeweils ersten Druck auf eine Maustaste. Bei vielen interaktiven Windows-Programmen kann der Anwender aber eine Maustaste drücken und die Maus dann bewegen, um Linien oder Rechtecke zu zeichnen oder um Grafiken zu verschieben. Wir wollen nun auch in diesem Beispielprogramm Mausbewegungen abfangen und auf sie durch das Zeichnen von Linien reagieren.

Wir tun dies durch das Beantworten weiterer Botschaften. Windows liefert die Botschaft *WM_MOUSEMOVE*, wenn der Anwender die Maus bewegt, und *WM_LBUTTONUP*, wenn er die linke Maustaste losläßt.

Normalerweise empfängt das Fenster eine *WM_LBUTTONDOWN*-Botschaft, darauf folgt eine Reihe von *WM_MOUSEMOVE*-Botschaften (eine für jeden Punkt, über den die Maus bewegt wird). Schließlich folgt noch eine *WM_LBUTTONUP*-Botschaft.

Ein typisches Windows-Grafikprogramm antwortet auf *WM_LBUTTONDOWN* durch die Initiierung eines Zeichenvorgangs (dazu gehört unter anderem auch die Anforderung eines Bildschirmkontexts). *WM_MOUSEMOVE*-Botschaften werden durch das Zeichnen oder Verschieben von Grafiken und *WM_LBUTTONUP* durch die Beendigung des Zeichenvorgangs (einschließlich der Freigabe des Bildschirmkontexts) beantwortet.

Die folgende Tabelle faßt die gebräuchlichsten Botschaften infolge von Mausereignissen zusammen.

Tabelle 3.1 Botschaften für Mausereignisse

Botschaft	Ereignis
WM_LBUTTONDOWN	Anwender drückt auf die linke Maustaste.
WM_RBUTTONDOWN	Anwender drückt auf die rechte Maustaste.
WM_MOUSEMOVE	Anwender bewegt die Maus.

Botschaft	Ereignis
WM_LBUTTONUP	Anwender läßt die linke Maustaste los.
WM_RBUTTONUP	Anwender läßt die rechte Maustaste los.
WM_LBUTTONDBLCLK	Anwender klickt mit der linken Maustaste doppelt.
WM_RBUTTONDBLCLK	Anwender klickt mit der rechten Maustaste doppelt.

Mausbewegungs-Botschaften beantworten

Die folgende Tabelle zeigt, wie auf Botschaften, die das Programm durch Bewegen der Maus erhält, reagiert werden muß, wenn ein Programm zum Linienzeichnen erstellt werden soll.

Tabelle 3.2 Die in Schritt 4 verwendeten Botschaften

Botschaft	Antwort
WM_LBUTTONDOWN	Löschen des Bildschirms, Anfordern und Speichern eines Bildschirmkontexts in *DmgDc*. Plazieren des Zeichenstifts auf dem angeklickten Punkt.
WM_MOUSEMOVE	Zeichnen einer Linie vom vorherigen zum aktuellen Punkt, solange die Maustaste gedrückt ist.
WM_LBUTTONUP	Freigabe von *DragDc*.

Wie oben bereits ausgeführt, folgt auf *WM_LBUTTONDOWN* immer *WM_LBUTTONUP*. Dazwischen können (müssen aber nicht) *WM_MOUSEOVE*-Botschaften übergeben werden. Auf jeden Fall müssen Sie einen Bildschirmkontext abrufen, wenn Ihr Fenster eine *WM_LBUTTONDOWN*-Botschaft empfängt. Dieser Kontext muß auf eine *WM_LBUTTONUP*-Botschaft hin wieder freigegeben werden. In unserem Fall wird für alle Zeichenvorgänge, die während des Drückens der linken Maustaste stattfinden, nur ein einziger Bildschirmkontext verwendet.

Die Kombination einer Anforderung eines Bildschirmkontexts mit seiner Freigabe ist wichtig für die einwandfreie Funktion von Grafikprogrammen. Es ist deshalb vorteilhaft, eine weitere Sicherung einzubauen. Definieren wir dazu ein neues *BOOL*-Datenelement für *TMyWindow* (die Hauptfensterklasse) mit dem Namen *ButtonDown*. *WMLButtonDown* setzt dieses Element auf *TRUE*, *WMLButtonUp* setzt es auf *FALSE*. Der Wert von *ButtonDown* kann dann vor dem Anfordern und der Freigabe des Bildschirmkontexts überprüft werden.

Die drei Elementfunktionen für die Beantwortung der durch Mausbewegungen hervorgerufenen Botschaften sehen folgendermaßen aus:

```
void TMyWindow::WMLButtonDown(RTMessage Msg)
{
   InvalidateRect(HWindow, NULL, TRUE);
   if ( !ButtonDown )
```

```
    {
      ButtonDown = TRUE;
      SetCapture(HWindow);
      DragDC = GetDC(HWindow);
      MoveTo(DragDC, Msg.LP.Lo, Msg.LP.Hi);
    }
}

void TMyWindow::WMMouseMove(RTMessage Msg)
{
   if ( ButtonDown )
     LineTo(DragDC, Msg.LP.Lo, Msg.LP.Hi);
}

void TMyWindow::WMLButtonUp(RTMessage)
{
   if ( ButtonDown )
   {
     ButtonDown = FALSE;
     ReleaseCapture();
     ReleaseDC(HWindow, DragDC);
   }
}
```

Hinweis: Dieses Programmfragment finden Sie in STEP4.CPP.

MoveTo und *LineTo* sind Grafikfunktionen der Windows-API, die die aktuelle Zeichenposition verschieben bzw. eine Linie zur aktuellen Position zeichnen. Sie benötigen für die korrekte Ausführung ein Handle für den Bildschirmkontext, den *TMyWindow* in seinem Datenelement *DragDc* speichert. (Denken Sie immer daran, daß Sie nicht direkt im Fenster, sondern im zugehörigen Bildschirmkontext zeichnen.)

Wird die Maus aus *TMyWindow* hinausbewegt, sendet Windows die *WM_MOUSEMOVE*-Botschaften weiterhin an das Hauptfenster, auch dann, wenn die Maus in ein benachbartes Fenster bewegt wurde. Dies ist das Ergebnis davon, daß *TMyWindow* Eingaben der Maus über einen Aufruf von *SetCapture* abfängt. Das Abfangen aller Mauseingaben wird mit einem Aufruf von *ReleaseCapture* freigegeben.

Die Objektdefinition für *TMyWindow* muß mit dem jeweiligen Elementfunktions-Header für *WMMouseMove* und *WMLButtonUp* aktualisiert werden:

```
virtual void WMLButtonUp(RTMessage Msg) =
  [WM_FIRST + WM_LBUTTONUP];
virtual void WMMouseMove(RTMessage Msg) =
  [WM_FIRST + WM_MOUSEMOVE];
```

Der momentane Stand der Steps-Anwendung ist in der Datei STEP4.CPP enthalten.

Schritt 5: Die Linienstärke ändern

Bis zum jetzigen Zeitpunkt können mit der Steps-Anwendung nur dünne Linien gezeichnet werden. Grafikprogramme ermöglichen aber normalerweise die Änderung der Linienstärke. Um unserem Programm diese Eigenschaft zu verleihen, ändern wir in Wirklichkeit nicht die Stärke der Linien, sondern die des Stiftes, mit dem die Linien gezeichnet werden. Stifte gehören neben Pinseln, Schriften und Paletten zu den Zeichenwerkzeugen, die der Bildschirmkontext zur Verfügung stellt und verwaltet. Indem wir der Steps-Anwendung die Fähigkeit zum Setzen einer neuen Linienstärke geben, lernen wir auch, wie einem Bildschirmkontext neue Zeichenwerkzeuge zugeordnet werden können.

Der Anwender soll die Stärke der Linien in einem Eingabefeld (der Klasse *TInputDialog*) eingeben können. Abbildung 3.2 zeigt, wie dieses Eingabefeld aussieht.

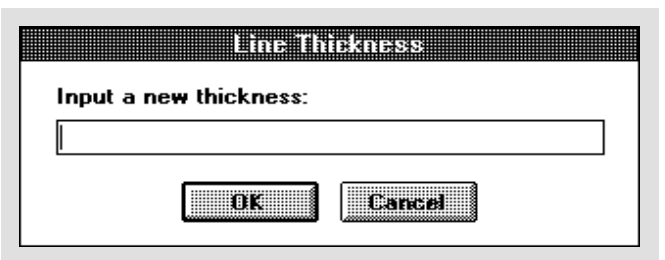


Abbildung 3.2
Die Linienstäreke ändern

Einen neuen Stift auswählen

Um die Linienstärke ändern zu können, sind etwas detailliertere Kenntnisse der Windows-Grafik und des Bildschirmkontexts erforderlich, auf die wir kurz eingehen wollen.

Folgende Zeichenwerkzeuge stehen zur Erstellung von Grafiken und Text in einem Fenster zur Verfügung: Stifte, Pinsel und Schriften. Die Beschreibungen dieser Zeichenelemente stehen (ähnlich wie bei anderen Schnittstellenelementen) im Speicherbereich von Windows. Für die Identifizierung beider Elementtypen werden *Handle* verwendet. Da Zeichenwerkzeuge keine ObjectWindows-Objekte sind, müssen die entsprechenden Objekte von Ihren Programmen erzeugt und später wieder aus dem Windows-Speicher entfernt werden. Wenn Sie das Entfernen der Zeichenwerkzeuge vergessen, verschwenden Sie Windows-Speicher!

Ein Zeichenwerkzeug ist im Grunde nichts anderes als der Pinsel eines Malers, während der Bildschirmkontext die Leinwand darstellt. Nach dem »Bereitstellen« der Zeichenwerkzeuge (Pinsel) und des Bildschirmkontexts (der Leinwand) erfolgt die Auswahl des benötigten Zeichenwerkzeugs. Entsprechend wird auch in einem Windows-Programm das korrekte Zeichenwerkzeug für einen Bildschirmkontext gewählt. Wie ist es nun möglich, daß Sie Text und Linien in Ihren

Fenstern zeichnen können, ohne Zeichenwerkzeuge ausgewählt zu haben? Zu jedem Bildschirmkontext gehört ein Standardsatz von Zeichenwerkzeugen, die nicht explizit ausgewählt werden müssen: ein dünner schwarzer Stift, ein kompakter schwarzer Pinsel und eine Systemschrift. Wir benötigen jedoch für unser Beispiel einen anderen, etwas stärkeren Stift für das Zeichnen im Fenster.

Der erste Schritt besteht darin, der Steps-Anwendung die Anzeige von Eingabefeldern zu ermöglichen. Dazu binden Sie die Header-Datei inputdia.h in die .CPP-Quelldatei ein. Für Eingabefelder muß eine Dialogressource in einer .RC-Datei definiert sein. Die hier benötigte Dialogressource ist in INPUTDIA.DLG definiert. Binden Sie diese Datei in Ihre .RC-Datei ein. STEPS.RC enthält alle für die Steps-Anwendung erforderlichen Ressourcendefinitionen. Zu diesem Zeitpunkt muß lediglich folgendes in der .RC-Datei stehen:

```
#include "windows.h"
#include "owlrc.h"
rcinclude inputdia.dlg
```

Jetzt benötigen wir ein Datenelement für *TMyWindow*, in dem ein Handle für den gewünschten Stift gespeichert werden kann. In unserem Programm beschränken wir uns auf das Zeichnen und Anzeigen von Linien mit jeweils nur einer Linienstärke. Der Stift, der dieser Stärke entspricht, wird in einem neuen *TMyWindow*-Datenelement namens *ThePen* gespeichert. Außerdem wird eine Elementfunktion mit dem Namen *SetPenSize* benötigt, die die Erzeugung des neuen Stiftes und das Löschen des alten übernimmt.

Die Klassendeklaration von *TMyWindow* sieht dann wie folgt aus:

```
_CLASSDEF(TMyWindow)

class TMyWindow : public TWindow
{
public:
  HDC DragDC;
  BOOL ButtonDown;
  HPEN ThePen;
  int PenSize;
  TMyWindow(PTWindowsObject AParent, LPSTR ATitle);
  ~TMyWindow();
  virtual BOOL CanClose();
  void SetPenSize(int NewSize);
  virtual void WMLButtonDown(RTMessage Msg) =
    [WM_FIRST + WM_LBUTTONDOWN];
  virtual void WMLButtonUp(RTMessage Msg) =
    [WM_FIRST + WM_LBUTTONUP];
  virtual void WMMouseMove(RTMessage Msg) =
    [WM_FIRST + WM_MOUSEMOVE];
  virtual void WMRButtonDown(RTMessage Msg) =
    [WM_FIRST + WM_RBUTTONDOWN];
};
```

Hinweis: Dieses Programmfragment finden Sie in STEP5.CPP.

Zur Initialisierung dieser neuen Datenelemente modifizieren wir nun den Konstruktor, so daß er den Stift einrichtet, und definieren den Destruktor, damit er den Stift freigibt.

```
TMyWindow::TMyWindow(PTWindowsObject AParent, LPSTR ATitle)
  : TWindow(AParent, ATitle)
{
  ButtonDown = FALSE;
  PenSize = 1;
  ThePen = CreatePen(PS_SOLID, PenSize, 0);
}

TMyWindow::~TMyWindow()
{
  DeleteObject(ThePen);
}
```

Hinweis: Dieses Programmfragment finden Sie in STEP5.CPP.

Wir ändern nun die Elementfunktion *WMLButtonDown* dahingehend, daß sie den aktuellen Stift (*ThePen*) für den neu angeforderten Bildschirmkontext auswählt.

Wie *MoveTo* und *MessageBox* ist auch *SelectObject* eine Funktion der Windows-API.

```
void TMyWindow::WMLButtonDown(RTMessage Msg)
{
  InvalidateRect(HWindow, NULL, TRUE);
  if ( !ButtonDown )
  {
    Buttondown = TRUE;
    SetCapture(HWindow);
    DragDC = GetDC(HWindow);
    SelectObject(DragDC, ThePen);
    MoveTo(DragDC, Msg.LP.Lo, Msg.LP.Hi);
  }
}
```

Hinweis: Dieses Programmfragment finden Sie in STEP5.CPP.

Den Stift ändern

Diese Elementfunktion wählt den bereits erzeugten Stift für den Bildschirmkontext. Die tatsächliche Erzeugung des Stifts erfolgt mit der Elementfunktion *SetPenSize*:

```
void TMyWindow::SetPenSize(int NewSize)
{
  DeleteObject(ThePen);
  ThePen = CreatePen(PS_SOLID, NewSize, 0);
  PenSize = NewSize;
}
```

Hinweis: Dieses Programmfragment finden Sie in STEP5.CPP.

Der Aufruf der Windows-Funktion *CreatePen* ist eine Möglichkeit zur Erzeugung eines Windows-Zeichenstifts mit einer bestimmten Stärke. In *ThePen* wird ein Handle für den Stift gespeichert. Das Löschen des vorherigen Stiftes ist wichtig, da andernfalls der Speicher von Windows bald erschöpft wäre.

Ein Eingabefeld verwenden

Das Eingabefeld zur Änderung der Linienstärke soll durch den Druck der rechten Maustaste eingeblendet werden. Dazu müssen wir die Elementfunktion *WMRButtonDown* so überschreiben, daß sie ein Eingabefeld anzeigt. Wir verwenden ein Eingabefeld, das in ObjectWindows vordefiniert ist. Es handelt sich um ein einfaches Dialogfenster mit einer Zeile für die Texteingabe.

Nachdem das Objekt des Eingabefeldes erstellt wurde, kann es durch einen Aufruf von *TModule::ExecDialog* als modales Dialogfenster benutzt werden. Die Verwendung von *ExecDialog* entspricht dem Einsatz der Elementfunktion *MakeWindow,* die für Fensterobjekte aufgerufen wird. Beide erzeugen nur dann das Schnittstellenobjekt, wenn dieses gültig ist. *ExecDialog* endet aber erst, wenn der Anwender das Dialogfenster durch Anklicken von *OK* oder *Abbrechen* schließt. Klickt er *OK* an, wird *InputText* mit der Eingabe des Anwenders gefüllt. Da eine Zahl für die Linienstärke benötigt wird, muß der Text in eine Zahl konvertiert und in einem Aufruf von *SetPenSize* übergeben werden. Bei jeder Auswahl einer neuen Linienstärke durch den Anwender muß der alte Stift gelöscht und ein neuer erzeugt werden.

```
void TMyWindow::WMRButtonDown(RTMessage)

{
  char InputText[6];
  int NewPenSize;

  sprintf(InputText, "%d", PenSize);

  if ( GetApplication()->ExecDialog( new TInputDialog(this,
       "Line Thickness", "Input a new thickness:",
       InputText, sizeof InputText)) == IDOK )
  {
       NewPenSize = atoi(InputText);
       if ( NewPenSize < 0 )
         NewPenSize = 1;
       SetPenSize(NewPenSize);
  }

}
```

Der momentane Stand der Steps-Anwendung ist in der Datei STEP5.CPP enthalten.

Schritt 6: Grafiken zeichnen

Wahrscheinlich sind Sie überrascht, daß Text und Grafiken, die mit den Windows-Funktionen *TextOut* und *LineTo* in ein Fenster gezeichnet wurden, beim Verkleinern, Vergrößern oder beim Sichtbarmachen des Fensters verschwinden. Warum ist das so?

Interessanter ist eigentlich die Frage: Wo waren Text und Grafik vorher? Bisher wurden Text oder Linien nicht in Variablen gespeichert. Nachdem Grafikdaten über Funktionsaufrufe an Windows übergeben wurden, ist es nicht möglich, für eine Neuanzeige des Bildschirms auf sie zuzugreifen. Um den Zugriff zu ermöglichen, müssen die Grafikdaten irgendwie gespeichert werden. Ein Objekt eignet sich vorzüglich für diese Aufgabe. Es kann einfache und komplizierte Grafiken speichern. Diese können also problemlos als Datenelemente des Hauptfensters verwaltet werden.

Das Zeichen-Modell

Windows sendet eine *WM_PAINT*-Botschaft an Ihre Anwendung, wenn die Anzeige eines Ihrer Fenster aktualisiert (*neu gezeichnet*) werden muß. Eine *WM_PAINT*-Botschaft wird z.B. gesendet, wenn der Anwender eines der Fenster in der Größe verändert oder verbirgt bzw. wieder sichtbar macht. ObjectWindows fängt die *WM_PAINT*-Botschaft ab, und ruft die *WMPaint*-Elementfunktion des Fensters auf.

TWindowsObject definiert eine *WMPaint*-Elementfunktion, die ihrerseits die virtuelle Funktion *Paint* aufruft, nachdem sie *BeginPaint* und gegebenenfalls *BeginView* ausgeführt hat, um das Fenster auf die Zeichnen-Operationen vorzubereiten. Nach dem Aufruf von *Paint* gibt sie den Bildschirmkontext wieder frei. Überschreiben Sie *Paint* in von *TWindow* abgeleiteten Klassen, damit Ihre Fensteranzeige neu gezeichnet wird.

Es gibt einen großen Unterschied zwischen dem Zeichnen von Grafiken mit der *Paint*-Elementfunktion und den anderen bisher verwendeten Funktionen (zum Beispiel Zeichnen als Antwort auf ein Mausereignis). *Paint* verfügt bereits über einen Bildschirmkontext. Er wird im Parameter *PaintDC* übergeben, das Programm braucht ihn deshalb nicht anzufordern oder freizugeben. Allerdings müssen die Zeichenwerkzeuge für *PaintDC* neu ausgewählt werden.

Für das Zeichnen des Fensterinhalts führen wir die Schritte noch einmal aus, die für die Erstellung der ursprünglichen Zeichnung in *DragDC* erforderlich waren, verwenden aber nun *PaintDC* anstelle von *DragDC*. Das Ergebnis auf dem Bildschirm ist das gleiche wie beim ersten Zeichnen durch den Anwender; vergleichbar ist der Vorgang mit dem Wiederabspielen eines aufgenommenen Konzerts. Ist es nun »live« oder ist es ObjectWindows? Zuerst müssen die Grafikdaten, die für die *Paint*-Elementfunktion notwendig sind, als Objekte gespeichert werden.

Grafiken als Objekte speichern

In Schritt 4 haben Sie gesehen, daß die Linie ursprünglich als Antwort auf die *WM_MOUSEMOVE*-Botschaften gezeichnet wurde. Diese Botschaften enthalten die aktuelle Mausposition in Form von x- und y-Koordinaten eines Fensterpunktes. Die Linie ist dadurch entstanden, daß die von den Botschaften gesendeten Punkte miteinander verbunden wurden. Diese Punkte sind diejenigen Grafikdaten, die für das erneute Zeichnen der Linie erforderlich sind.

Für die Repräsentation dieser Punkte verwenden wir *TPoint*, eine von *Object* abgeleitete Klasse. An diesem Punkt kann eine einfache Struktur anstelle eines Objekts eingesetzt werden. Wir speichern diese *TPoint*-Objekte in einer der Container-Klassenbibliotheken, die alle von *Object* abgeleitete Objekte erfordern. In späteren Arbeitsschritten werden wir Elementfunktionen zu *TPoint* hinzufügen.

Wir speichern die *TPoint*-Objekte in einem *TMyWindow*-Datenelement mit dem Namen *Points*. *Points* ist eine Instanz vom Typ *TPointArray*, die von der Container-Klasse *Array* abgeleitet wird. Der Grund dafür, daß wir anstelle von *Array TPointArray* verwenden, liegt darin, daß wir *TPointArray* neue, in *Array* nicht vorhandene Eigenschaften hinzufügen wollen. Da *Array*-Objekte (und von *Array* abgeleitete Klassen) die Fähigkeit besitzen, dynamisch zu wachsen, eignen sie sich sehr gut für das Speichern einer unbekannten Zahl von Punkten. *Array*-Objekte und die anderen Container-Klassen sind in der Borland-C++-Dokumentation ausführlich beschrieben. Im folgenden sehen Sie einen Teil der *TPoint*-Definition:

```
class TPoint: public Object
{
public:
  int X, Y;
  TPoint(int AX, int AY) {X = AX, Y = AY;}
.
.
.
};
```

Im folgenden sehen Sie einen Teil der *TPointArray*-Deklaration:

```
_CLASSDEF(TPointArray)
class TPointArray : public Array
{
  TPointArray(int upper, int lower = 0, sizeType aDelta = 0)
    : Array(upper, lower, aDelta){};
  void DeleteAll();
.
.
.
};
```

Hinweis: Dieses Programmfragment finden Sie in STEP6.CPP.

Sehen Sie sich nun die aktualisierte *TMyWindow*-Deklaration an, in die das Datenelement *Points* eingefügt wurde:

```
_CLASSDEF(TMyWindow)
class TMyWindow : public TWindow
{
.
.
.
  PTPointArray Points;
.
.
.
};
```

Hinweis: Dieses Programmfragment finden Sie in STEP6.CPP.

Achten Sie unbedingt darauf, daß *Points* im *TMyWindow*-Konstruktor erstellt und im *TMyWindow*-Destruktor freigegeben wird.

```
TMyWindow::TMyWindow(PTWindowsObject AParent, LPSTR ATitle)
  : TWindow(AParent, ATitle)
{
  ButtonDown = FALSE;
  PenSize = 1;
  ThePen = CreatePen(PS_SOLID, PenSize, 0);
  Points = new TPointArray(50, 0, 50);
}
TMyWindow::~TMyWindow()
{
  delete Points;
  DeleteObject(ThePen);
}
```

Hinweis: Dieses Programmfragment finden Sie in STEP6.CPP.

Zum Anlegen von *Points* müssen wir *WMLButtonDown* und *WMMouseMove* modifizieren.

```
void TMyWindow::WMLButtonDown(RTMessage Msg)
{
  Points->DeleteAll();
  InvalidateRect(HWindow, NULL, TRUE);
  if ( !ButtonDown )
  {
    ButtonDown = TRUE;
    SetCapture(HWindow);
    DragDC = GetDC(HWindow);
    SelectObject(DragDC, ThePen);
    MoveTo(DragDC, Msg.LP.Lo, Msg.LP.Hi);
    Points->add(* (new TPoint(Msg.LP.Lo, Msg.LP.Hi)));
  }
}
```

```
void TMyWindow::WMMouseMove(RTMessage Msg)
{
  if ( ButtonDown )
  {
    LineTo(DragDC, Msg.LP.Lo, Msg.LP.Hi);
    Points->add(* (new TPoint(Msg.LP.Lo, Msg.LP.Hi)));
  }
}
```

WMLButtonUp muß nicht geändert werden. In *TMyWindow::WMLButtonDown* werden vor dem Zeichnen einer neuen Linie alle *TPoint*-Punkte aus dem Array *Points* entfernt. Dies geschieht durch den Aufruf von *DeleteAll*():

```
void TPointArray::DeleteAll()
{
 RArrayIterator PointIterator = (RArrayIterator)initIterator();
 while ( int(PointIterator) != 0 )
 {
   RObject AnObject = PointIterator++;
   if ( AnObject != NOOBJECT )
     detach(AnObject, 1);   //detach and delete
 }
 delete &PointIterator;
}
```

Hinweis: Dieses Programmfragment finden Sie in STEP6.CPP.

Gespeicherte Grafiken neu zeichnen

Nach dem Anlegen von *Points* müssen Sie jetzt Programmtext schreiben, der ein *TMyWindow* zeichnet, indem die Linien zwischen den Punkten erneut gezogen werden. Wir schreiben eine *Paint*-Elementfunktion für *TMyWindow*, die die Schritte mit Hilfe des *Points*-Datenelements in ihren *WMLButtonDown*- und *WMMouseMove*-Elementfunktionen ausführt. Im folgenden sehen Sie die *Paint*-Elementfunktion:

```
void TMyWindow::Paint(HDC DC, PAINTSTRUCT&)
{
  RArrayIterator PointIterator =
                (RArrayIterator)(Points->initIterator());
  BOOL First = TRUE;

  SelectObject(DC, ThePen);
  while ( int(PointIterator) != 0 )
  {
    RObject AnObject = PointIterator++;
    if ( AnObject != NOOBJECT )
    {
```

```
        if ( First )
        {
          MoveTo(DC, ((PTPoint)(&AnObject))->X,
                     ((PTPoint)(&AnObject))->Y);
          First = FALSE;
        }
        else
          LineTo(DC, ((PTPoint)(&AnObject))->X,
                     ((PTPoint)(&AnObject))->Y);
      }
    }
    delete &PointIterator;
  }
```

Das Hauptfenster der Steps-Anwendung speichert einen Zeiger auf ein *Array* von *TPoint* in seinem Datenelement *Points*. Wenn der Anwender eine Linie zeichnet, empfängt das Fenster *WM_MOUSEMOVE*-Botschaften, die die Mauspositionen enthalten. Sie müssen *TPoint*-Objekte erstellen, die die Positionsdaten aufnehmen, und jedes *TPoint*-Objekt in das *Point*-Array einfügen. Wenn dann das Fenster erneut gezeichnet werden muß, können Sie die Linie durch das Verbinden der *TPoint*-Objekte wieder darstellen.

4. Ein Menü hinzufügen

Die meisten Windows-Anwendungen enthalten im Hauptfenster ein Menü, das dem Anwender eine Reihe von Auswahlpunkten zur Verfügung stellt (beispielsweise *Datei/Speichern*, *Datei/Öffnen* oder *Hilfe*). In diesem Kapitel werden wir für die Steps-Anwendung ein derartiges Menü erstellen.

In einer fensterorientierten Umgebung fällt die Auswahl aus einem Menü in die gleiche Kategorie wie ein Klicken mit der Maus. In beiden Fällen handelt es sich um ein Ereignis, das vom Anwender hervorgerufen wird. Die Antwort auf die Auswahl eines Menüpunkts erfolgt im wesentlichen so wie die Beantwortung anderer Anwendereingaben. Sie umfaßt folgende Schritte:

1. Entwurf des Menüs als Menüressource.
2. Laden der Menüressource in das Hauptfensterobjekt.
3. Definition von Antworten auf Menüauswahlen.
4. Compilieren der Anwendung zu einer ausführbaren Datei.
5. Hinzufügen der Menüressource zur ausführbaren Datei.

Menüressourcen

In einer Anwendung, die über ein Menü verfügt, muß eine Menüspezifikation mit dem Text der Auswahlpunkte und der Struktur der Haupt- und Unterpunkte vorhanden sein. Viele Anwendungen verfügen über Tastenkürzel für Menüoptionen (die sog. *Akzeleratoren* oder *Tastenkürzel*). Der Anwender kann dann anstelle *Bearbeiten/Einfügen* mit der Maustaste auswählen, Umschalt Einfg drücken und damit denselben Effekt erzielen. Menüs und Tastenkürzel sind keine Bestandteile des C++-Quelltexts. Sie gehören zu einer separaten Spezifikation, die als *Ressource* bezeichnet wird. Windows speichert Ressourcen in einer kompakten Art und Weise ab.

Eine Anwendung greift auf ihre zugehörigen Ressourcen mit Hilfe einer Ressourcen-ID zu. Es kann sich dabei um einen Integerwert (zum Beispiel 100), oder einen String-Bezeichner (zum Beispiel »*MyMenu*«) handeln. Die Unterscheidung der einzelnen Menüauswahlen erfolgt durch die Menü-ID.

Mit dem Ressourceneditor oder dem Ressourcencompiler wird eine Menüressource erzeugt. Die Datei STEPS.RC enthält die Menüressource der StepsAnwen-

dung im Quelltextformat. Wenn Sie die Borland-C++-IDE verwenden, können Sie STEPS.RC in das Projekt der Steps-Anwendung einfügen. Die IDE übersetzt die Datei dann mit Hilfe des Ressourcencompilers (RC). STEPS.RC sieht folgendermaßen aus:

```
#include <windows.h>
#include <owlrc.h>
#include "steps.h"

rcinclude inputdia.dlg
rcinclude filedial.dlg

COMMANDS MENU LOADONCALL MOVEABLE PURE DISCARDABLE
BEGIN
  POPUP "&File"
  BEGIN
    MenuItem "&New", CM_FILENEW
    MenuItem "&Open", CM_FILEOPEN
    MenuItem "&Save", CM_FILESAVE
    MenuItem "Save&As", CM_FILESAVEAS
  END
  MenuItem "&Help", CM_HELP
END
```

Die Abbildung 4.1 zeigt dieses Menü (Ressourcen-ID "COMMANDS"), das Help-Menü (Menü-ID *CM_HELP*) und den Auswahlpunkt *File* mit den Unterpunkten *New*, *Open*, *Save* und *SaveAs* (Menü-ID *CM_FILENEW*, *CM_FILEOPEN*, *CM_FILESAVE* und *CM_FILESAVEAS*). Hauptmenüpunkte, zu denen Unterpunkte existieren (wie z.B. *File*), haben keine Menü-IDs, ihre Auswahl hat nur die Anzeige der Unterpunkte zur Folge. Hat ein Hauptmenüpunkt keine Unterpunkte (wie beispielsweise *Help*), verfügt er über eine Menü-ID. Seine Auswahl führt zur unmittelbaren Ausführung einer Aktion.

Abbildung 4.1 Die Steps-Anwendung mit einer Menüressource

Da das Menü außerhalb von ObjectWindows entworfen wurde und der Umgang mit dem Menü ziemlich einfach ist, muß für das Menü kein neues

Objekt angelegt werden. Stattdessen wird es zu einem Attribut des Hauptfensters (wie es zum Beispiel auch der Titel des Hauptfensters ist). Im nächsten Schritt wird das Menü mit dem Hauptfenster verknüpft.

Schritt 7: Ein Menü für das Hauptfenster

Wie erwähnt, ist das Menü einer Anwendung kein separates Objekt, das dem Hauptfenster zugehörig ist. Das Menü ist überhaupt kein Objekt, sondern ein Attribut des Hauptfensters. Es wird im *Menu*-Feld von *Attr* gespeichert, einem Fensterobjekt-Datenelement, das die *Erstellungsattribute* enthält. Das Menüattribut wird neben anderen durch den Aufruf von *AssignMenu* im Konstruktor für das Fenster gesetzt.

Die in STEPS.RC gespeicherte Menüressource hat die ID "COMMANDS". Der Aufruf von *AssignMenu* sieht folgendermaßen aus:

```
AssignMenu("COMMANDS");
```

Die Ressourcen-ID kann auch ein Integer sein. In diesem Fall kann auch *AssignMenu* aufgerufen werden, da die Funktion überladen ist, z.B.:

```
AssignMenu(100);
```

AssignMenu ist eine Elementfunktion von *TWindow*.

Der *TMyWindow*-Konstruktor sollte das im folgenden dargestellte Aussehen haben. Beachten Sie, daß der *TWindow*-Konstruktor aufgerufen wird, um die erforderliche Initialisierung aller Fensterobjekte durchzuführen.

```
TMyWindow::TMyWindow(PTWindowsObject AParent, LPSTR ATitle)
  : TWindow(AParent, ATitle)
{
  AssignMenu("COMMANDS");
  ButtonDown = FALSE;
  PenSize = 1;
  ThePen = CreatePen(PS_SOLID, PenSize, 0);
  Points = new TPointArray(50, 0, 50);
}
```

Wird nun das Hauptfenster angezeigt, enthält es das in Abbildung 4.1 gezeigte Menü. Allerdings zeigt die Auswahl eines Menüpunktes noch keine Wirkung. Dazu müssen wir die folgenden Schritte zum Beantworten der Menüauswahlen ausführen.

Eine Menübotschaft abfangen

Wenn der Anwender einen Menüpunkt auswählt oder das entsprechende Tastenkürzel verwendet, empfängt das Fenster eine Windows-*Befehlsbotschaft*.

Zur Verarbeitung einer bei einer Menüpunktauswahl gesendeten Befehlsbotschaft definieren wir eine *Befehlsantwortfunktion.* Wir verwenden dabei die folgende spezielle Funktionsdeklaration:

```
virtual void CMFileNew(RTMessage Msg) = [CM_FIRST + CM_FILENEW];
```

Hierbei ist *CM_FIRST* eine in ObjectWindows vordefinierte Konstante und eine Konstante, die die ID des entsprechenden Menüpunktes repräsentiert. In unserem Beispiel ist *CM_FILENEW* in owlrc.h definiert, so daß es nicht von Ihnen definiert werden muß. Wenn Sie jedoch eigene Menüeinträge definieren (so wie wir es für *Help* durchführen werden), können Sie eine Konstante wie `#define CM_HELP 201` definieren. Schreiben Sie alle Ressourcen-ID-Konstanten in eine Header-Datei, die Sie in Ihre .CPP- und .RC-Dateien einbinden können. Beachten Sie, daß die spezielle Konstante *CM_FIRST*-Erweiterung der Identifizierung einer Befehlsantwortfunktion dient. Verwechseln Sie diese nicht mit der Konstanten *WM_FIRST*, die Botschaftsantwortfunktionen für Fenster kennzeichnet. (Botschaftsbereiche und -Offsets werden in Kapitel 7 »Überblick über ObjectWindows« ausführlich beschrieben.)

Allen Funktionen zur Beantwortung von Botschaften oder Befehlen wird eine *TMessage*-Strukturreferenz (Typ *RTMessage*) übergeben. Diese Struktur wird jedoch nicht häufig in Befehlsantwortfunktionen verwendet. *Msg.WParam* enthält die ID des Menüs, das ausgewählt wurde. Normalerweise werden Sie *Msg.LP.Hi* ignorieren, das 1 enthält, wenn die Botschaft bei Verwendung des Tastenkürzels für den Menüeintrag gesendet wird. *Msg.LP.Lo* enthält immer Null.

Jetzt können alle Elementfunktionen zur Beantwortung von Befehlsbotschaften mit Hilfe der oben diskutierten Ressourcen-ID-Konstanten deklariert werden. (Beachten Sie, daß *CM_HELP* in steps.h und nicht in owlrc.h definiert ist.)

```
virtual void CMFileNew(RTMessage Msg) =
  [CM_FIRST + CM_FILENEW];
virtual void CMFileOpen(RTMessage Msg) =
  [CM_FIRST + CM_FILEOPEN];
virtual void CMFileSave(RTMessage Msg) =
  [CM_FIRST + CM_FILESAVE];
virtual void CMFileSaveAs(RTMessage Msg) =
  [CM_FIRST + CM_FILESAVEAS];
virtual void CMHelp(RTMessage Msg) =
  [CM_FIRST + CM_HELP];
```

Hinweis: Dieses Programmfragment finden Sie in STEP7.CPP.

Eine Menübotschaft beantworten

Es steht nun für jede Menüauswahl eine Elementfunktion zur Verfügung. Wählt der Anwender zum Beispiel *Help*, wird die *CMHelp*-Elementfunktion aufgerufen. Momentan wird in diesem Fall nur ein Meldungsfenster eingeblendet:

```
void TMyWindow::CMHelp(RTMessage)
{
  MessageBox(HWindow, "Feature not implemented", "Help", MB_OK);
}
```

Hinweis: Dieses Programmfragment finden Sie in STEP7.CPP.

Die folgende Abbildung 4.2 zeigt die Antwort der Steps-Anwendung auf die Auswahl von *Help*.

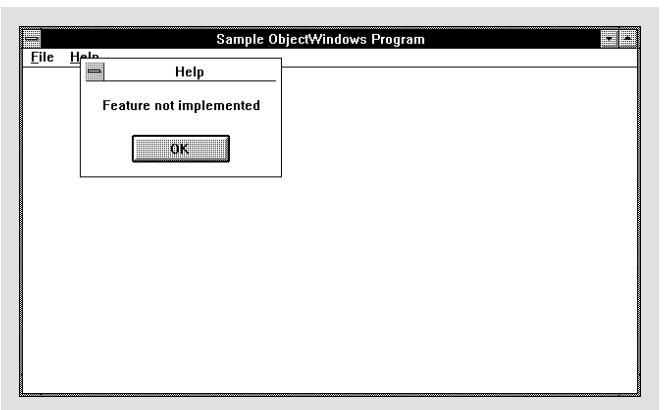


Abbildung 4.2
Das Hilfesystem der StepsAnwendung

An diesem Punkt können Sie auf die Menüauswahl von *File/New* mit dem Löschen des Bildschirms antworten. Fügen Sie folgende *CMFileNew*-Elementfunktion hinzu:

```
void TMyWindow::CMFileNew(RTMessage)
{
  Points->DeleteAll();
  InvalidateRect(HWindow, NULL, TRUE);
}
```

Hinweis: Dieses Programmfragment finden Sie in STEP7.CPP.

Die Elementfunktion löscht alle gespeicherten Punkte und erzwingt ein Neuzeichnen des Bildschirms. Da keine Punkte zum Neuzeichnen vorhanden sind, ist das Resultat ein leerer Bildschirm.

Für die Botschaften *CM_FILEOPEN*, *CM_FILESAVE* und *CM_FILESAVEAS* können ähnliche Dummy-Elementfunktionen wie *CMHelp* geschrieben werden. An späterer Stelle werden wir sie modifizieren, damit sie sinnvolle Aktionen ausführen.

Der momentane Quelltext der Steps-Anwendung ist in der Datei STEP7.CPP enthalten.

5. Einen Dialog erstellen

Ein richtiges Windows-Programm verfügt meist über viele verschiedene Fensterobjekte (Fenster, Steuerelemente und Dialoge), die ihm zugeordnet sind. Mit Ausnahme des Hauptfensters hat jedes Fenster, das untergeordnetes Fenster genannt wird, genau ein übergeordnetes Fenster, von dem es abhängig ist. Ein übergeordnetes Fenster kann viele, von ihm abhängige untergeordnete Fenster haben.

Aus diesen Beziehungen ergibt sich für jede Anwendung ein Netz von übergeordneten und untergeordneten Fenstern, wobei jeweils das Hauptfenster als oberstes übergeordnetes Fenster fungiert. Die anderen Fenster sind einem Fenster untergeordnet und haben ihrerseits eventuell wieder untergeordnete Fenster.

Die folgende Abbildung 5.1 zeigt die – bisher nur teilweise implementierten – Beziehungen zwischen übergeordneten und untergeordneten Fenstern der Steps-Anwendung.

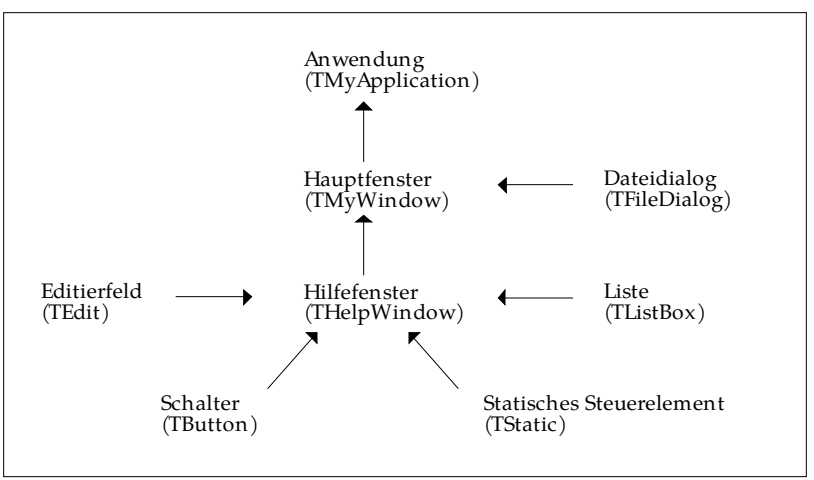


Abbildung 5.1 Eine Gruppe von über- und untergeordneten Fenstern

Untergeordnete Fenster lassen sich in zwei Gruppen einteilen. Ein Typ sind die sog. selbständigen untergeordneten Fenster. Diese Fenster steuern ihre

Anzeige und Position selbst. Meldungs- und Pop-up-Fenster gehören zu diesem Typ. Der zweite Typ sind die sog. unselbständigen untergeordneten Fenster. Diese Fenster haben eine feste Position im übergeordneten Fenster und werden mit diesem verschoben. Steuerelemente, wie zum Beispiel Schalter in Meldungsfenstern, gehören zu diesem Fenstertyp. Wir werden im folgenden Schritt zwei Typen von selbständigen untergeordneten Fensterobjekten erstellen: Pop-up-Fenster und Dialogfenster.

Über- und untergeordnete Fenster dürfen nicht mit Vorfahren und abgeleiteten Klassen verwechselt werden. Für über- und untergeordnete Fenster ist das *Zugehörigkeitsverhältnis* relevant. Beispielsweise gehört das Hilfefenster (das wir in Schritt 8 erstellen werden) zum Hauptfenster. Diese Art der Zugehörigkeit in Windows entspricht in etwa der Instanzenbindung der objektorientierten Programmierung.

Wir berücksichtigen dieses Prinzip in unseren Übungen, indem wir Objekte untergeordneter Fenster in den Datenelementen ihrer übergeordneten Fensterobjekte speichern. Diese Vorgehensweise ist zwar nicht unbedingt erforderlich, sie liefert jedoch klare, leicht zu wartende Programme.

Schritt 8: Ein Pop-up-Fenster hinzufügen

Als nächstes erstellen wir ein Pop-up-Fenster, das nach der Auswahl des Menüpunktes *Help* angezeigt wird, und das die Basis eines Hilfesystems für die Steps-Anwendung bildet. Das Pop-up-Fenster soll eine Instanz von *TWindow* werden. In Schritt 10 werden wir eine neue Klasse für das Fenster erzeugen und ihm eine Verhaltensweise zuordnen. Momentan beschränken wir uns auf seine Anzeige auf dem Bildschirm.

Da es sich bei dem Pop-up-Fenster um das erste neue Fenster handelt, das wir hinzufügen (das Hauptfenster wird ja automatisch angezeigt), müssen wir uns zuerst mit der Erstellung und Anzeige von Fensterobjekten und -elementen befassen.

Ein Pop-up-Fenster erzeugen und anzeigen

In Schritt 7 haben wir auf die Auswahl des Menüpunkts *Help* durch die Anzeige eines Meldungsfensters reagiert. Dieses Meldungsfenster ersetzen wir jetzt durch ein leeres Pop-up-Fensterobjekt mit dem Namen *HelpWindow*. In Schritt 10 werden wir die Verhaltensweisen für dieses leere Pop-up-Fenster definieren. Momentan beschränken wir uns darauf, *CMHelp* so zu überschreiben, daß das Hilfefenster angelegt wird:

```
void TMyWindow::CMHelp(RTMessage)
{
  PTWindow HelpWindow;
```

```
    HelpWindow = new TWindow(this, "Help System");
    HelpWindow->Attr.Style |= WS_POPUPWINDOW | WS_CAPTION;
    HelpWindow->Attr.X = 100;
    HelpWindow->Attr.Y = 100;
    HelpWindow->Attr.W = 300;
    HelpWindow->Attr.H = 300;
    GetApplication()->MakeWindow(HelpWindow);
}
```

Hinweis: Dieses Programmfragment finden Sie in STEP8.CPP.

In Kapitel 10 »Fensterobjekte« finden Sie im Abschnitt »Ein Fensterobjekt initialisieren« Informationen zur Erstellung von Fensterobjekten.

Die Funktion MakeWindow

Das Objekt *TModule* definiert eine wichtige Elementfunktion namens *MakeWindow*. *MakeWindow* führt alle Aktionen aus, die für die sichere Verknüpfung eines Schnittstellenelements mit einem Fensterobjekt erforderlich sind. »Sicher« bedeutet in diesem Fall, daß *MakeWindow* eine Prüfung auf Fehlerbedingungen durchführt, *bevor* es zu ernsthaften Problemen (beispielsweise einem Systemabsturz) kommen kann. *MakeWindow* führt zwei Aufgaben aus:

1. Die Funktion ruft *ValidWindow* auf, um das Schnittstellenobjekt auf seine Gültigkeit zu testen. *ValidWindow* überprüft, ob die Anwendung genügend Speicher zur Verfügung hat, um die Erstellung des Schnittstellenobjekts abzuschließen. Ist nicht genügend Speicherplatz vorhanden, kann die Erstellung nicht ausgeführt werden.

2. Nach der erfolgreichen Erstellung des Schnittstellenobjekts versucht *MakeWindow* mit einem Aufruf der *Create*-Elementfunktion des Objekts ein Fensterelement zu erstellen und zu verknüpfen. Über die Informationen in den Datenelementen des Objekts teilt *Create* Windows mit, welche Art von Schnittstellenelement erstellt werden soll. Kann das Element nicht erstellt werden, wird eine Fehlermeldung angezeigt.

Kann die Gültigkeitsprüfung oder die Erstellung des Schnittstellenelements nicht korrekt durchgeführt werden, entfernt *MakeWindow* das ungültige Fensterobjekt und liefert *NULL* zurück. Andernfalls liefert *MakeWindow* einen Zeiger auf das Schnittstellenelement zurück, das ihm als Parameter übergeben wurde.

Ein Aufruf von *MakeWindow* ist die sicherste Möglichkeit für die Erstellung von Schnittstellenelementen. Der direkte Aufruf von *Create* ohne die Überprüfung der Speicherkapazität und das Abfangen von Fehlern kann leicht zu einem Systemabsturz führen und ist deshalb nicht ratsam.

Ein Dialogfenster hinzufügen

Ein Dialogfenster ähnelt einem Pop-up-Fenster, bleibt aber normalerweise für kurze Zeit auf dem Bildschirm und führt eine – von einer Benutzereingabe abhängige – Aufgabe aus, beispielsweise die Auswahl eines Druckers. Das neue Dialogfenster für die Steps-Anwendung soll das Öffnen und Speichern von Dateien ermöglichen.

Wie ein Pop-up-Fenster ist auch ein Dialogfenster ein selbständiges untergeordnetes Fenster. Im Prinzip entspricht die Erstellung eines Dialogfensters als untergeordnetes Fenster dem Hinzufügen eines Pop-up-Fensters, das wir im letzten Abschnitt beschrieben haben. Ein Dialogfenster weist aber wichtige Unterschiede zu anderen Fenstern auf:

- Dialogklassen sind nicht von der Klasse *TWindow*, sondern von *TDialog* abgeleitet. Sowohl *TDialog* als auch *TWindow* sind von *TWindowsObject* abgeleitet.
- Dialogfenster benötigen normalerweise *Ressourcen*, die ihre Größe, Position und ihr Aussehen festlegen.
- Normalerweise führen Dialogfenster eine kurze Aufgabe aus und liefern einen Wert zurück. Beispielsweise liefert das *CanClose*-Meldungsfenster mit dem in Schritt 2 erstellten minimalen Dialog eine *Ja*- oder *Nein*-Antwort des Anwenders. Sehen Sie sich auch noch einmal das Dialogfenster für die Linienstärke in Schritt 5 an.

Unser nächster Schritt besteht im Hinzufügen eines in ObjectWindows vordefinierten Datei-Dialogfensters, das durch die Klasse *TFileDialog* definiert ist (vgl. Abbildung 5.2).

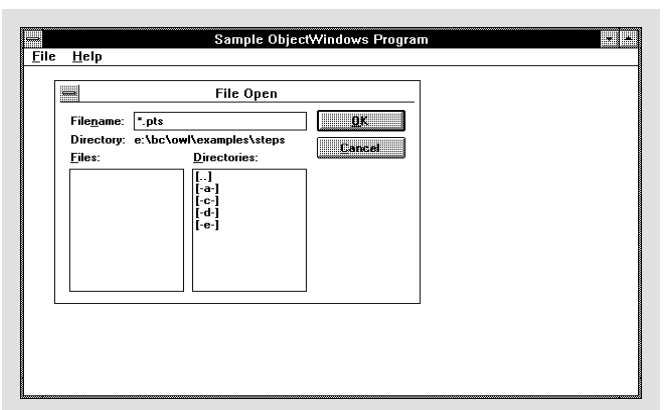


Abbildung 5.2 Die Steps-Anwendung mit dem DateiDialogfenster

Das Datei-Dialogfenster kann in jeder Situation eingesetzt werden, in der der Anwender eine Datei zum Laden oder Speichern auswählen soll. In einem Textverarbeitungsprogramm dient ein Datei-Dialogfenster beispielsweise zum Öffnen und Speichern von Dokumenten.

Im unserem Beispiel soll ein Datei-Dialogfenster als Reaktion auf die Auswahl von *File/Open* oder *File/Save* angezeigt werden. Das Datei-Dialogfenster ersetzt somit das Meldungsfenster mit der Anzeige »Feature not implemented«. In Schritt 9 werden wir es mit einigen Dateien verknüpfen, die geladen und gespeichert werden können. Jetzt wollen wir uns nur mit der Anzeige des Dialogfensters befassen.

Ein Datenelement hinzufügen

Das Objekt eines Datei-Dialogfensters wird nicht als Datenelement seines übergeordneten Fensters gespeichert. Stattdessen werden für alle benötigten Datei-Dialogfenster neue Objekte erstellt. Gespeichert werden nur die Daten, die mit dem Datei-Dialogfenster verwendet werden sollen: ein Dateiname. Das hat den Vorteil, daß keine Objekte mitzuführen sind, die sonst nirgends benötigt werden. Es werden nur die Daten gespeichert, die für die Initialisierung der jeweiligen Objekte nötig sind.

Für die Erstellung eines Datei-Dialogfensters werden drei Parameter benötigt: ein übergeordnetes Fenster, eine Ressourcen-ID und ein Dateiname. Die übergebene Ressourcen-ID ist einer der Bezeichner der Standard-Datei-Dialogfenster »open« oder »save« (*SD_FILEOPEN* oder *SD_FILESAVE*). Der ausgewählte Dateiname wird in dem übergebenen Dateinamenparameter zurückgeliefert. Dieser Parameter wird auch für die Übergabe eines Standard-Dateinamens für das Dialogfenster zum Speichern von Dateien verwendet. Er dient weiterhin zur Übergabe der Standard-Dateimaske für das Dialogfenster zum Öffnen von Dateien.

Die Deklaration von *TMyWindow* sieht nun folgendermaßen aus:

```
class TMyWindow : public TWindow
{
.
.
.
   char FileName[MAXPATH];
.
.
.
};
```

Einen Dialog führen

Abhängig von der an den *TFileDialog*-Konstruktor übergebenen Ressourcen-ID kann das Dialogfenster entweder zum Öffnen oder zum Speichern von Dateien eingesetzt werden. In beiden Fällen wird ein Dialogfenster ähnlich dem in Abbildung 5.2 gezeigten Fenster erzeugt.

Dialogfenster zum Öffnen und zum Speichern von Dateien unterscheiden sich in zwei Punkten:

1. Das Dialogfenster zum Öffnen enthält einen *OK*- und einen *Cancel*-Schalter, während das Dialogfenster zum Speichern von Dateien einen *Save*- und einen *Cancel*-Schalter beinhaltet.

2. Das Dialogfenster zum Speichern zeigt im Editierbereich seines Kombinationsfensters zu Beginn den aktuellen Dateinamen an.

CMFileOpen und *CMFileSaveAs* werden folgendermaßen umgeschrieben:

```
void TMyWindow::CMFileOpen(RTMessage)
{
  if ( GetApplication()->ExecDialog(new TFileDialog(this,
       SD_FILEOPEN, strcpy(FileName, "*.PTS"))) == IDOK )
     MessageBox(HWindow, FileName, "Open the file:", MB_OK);
}

void TMyWindow::CMFileSaveAs(RTMessage)
{
  if ( GetApplication()->ExecDialog(new TFileDialog(this,
       SD_FILESAVE, FileName)) == IDOK )
     MessageBox(HWindow, FileName, "Save the file:", MB_OK);
}
```

Hinweis: Dieses Programmfragment finden Sie in STEP8.CPP.

Der aktuelle Stand des Quelltextes der Steps-Anwendung ist in STEP8.CPP enthalten.

Zur »sicheren« Erstellung der Schnittstellenelemente haben wir *TApplication::MakeWindow* verwendet. Auf ähnliche Weise lassen sich durch Aufruf von *TApplication::ExecDialog* sichere Dialogfenster erzeugen und anzeigen. Wie *MakeWindow* ruft auch *ExecDialog ValidWindow* auf, um zu überprüfen, ob die Erstellung des Dialogobjekts erfolgreich war. Ist das der Fall, ruft *ExecDialog* die *Execute*-Elementfunktion des Dialogfensterobjekts auf. Nach einer fehlerlosen Ausführung liefert *ExecDialog* den von *Execute* zurückgegebenen Wert. Bei einem Fehler wird *IDCANCEL* zurückgeliefert. Für Ihr Programm entspricht dies der Auswahl von *Cancel* durch den Anwender. In beiden Fällen gibt *ExecDialog* das Objekt des Dialogfensters frei.

Ein Aufruf von *ExecDialog* ist die sicherste Möglichkeit zur Durchführung eines Dialogs. Der direkte Aufruf von *Execute* wäre zwar auch möglich, könnte aber zum Systemabsturz führen und ist deshalb nicht zu empfehlen.

Schritt 9: Die Zeichnung in einer Datei speichern

TMyWindow verfügt nun über eine Datenrepräsentation für seine aktuelle Linie (sein *Points*-Datenelement). Sie können diese Daten in eine Datei (genauer gesagt einen Datei-Stream) übertragen und von dort wieder einlesen. In Arbeits-

schritt 9 werden wir Datenelemente für das Speichern von Statusinformationen hinzufügen, und die Elementfunktionen ändern, mit deren Hilfe eine .PTS-Datei geöffnet und gespeichert werden kann.

Den Status überwachen

Zwei Eigenschaften der Zeichnung müssen im Auge behalten werden: »Muß die Datei gespeichert werden?« und »Ist momentan eine Datei geladen?«. Diese Fragen können als *BOOL*-Attribute von *TMyWindow* umgesetzt und als Datenelemente dargestellt werden:

```
class TMyWindow : public TWindow
{
.
.
.
  BOOL IsDirty, IsNewFile;
.
.
.
};
```

IsDirty ist *TRUE*, wenn die Datei gespeichert werden muß, da sie seit der letzten Änderung modifiziert wurde oder bisher noch gar nicht gespeichert worden ist. *IsDirty* muß auf *TRUE* gesetzt werden, wenn der Anwender zu zeichnen beginnt (*WMLButtonDown*). Öffnet der Anwender eine neue Datei oder speichert er die vorhandene, wird *IsDirty* auf *FALSE* gesetzt. Beendet er das Programm (*CanClose*), sollte der Status von *IsDirty* überprüft und, falls er *TRUE* ist, ein Meldungsfenster angezeigt werden.

IsNewFile ist nur dann *TRUE*, wenn das Programm das erste Mal angezeigt wird oder der Anwender das Menü *File/New* wählt (*CMFileNew*). Durch das Öffnen (*CMFileOpen*) oder das Speichern (*CMFileSave* oder *CMFileSaveAs*) einer Datei wird *IsNewFile* auf *FALSE* gesetzt. *CMFileSave* stellt mit Hilfe von *IsNewFile* fest, ob die Datei sofort gespeichert werden kann (*FALSE*) oder ob der Anwender einen Dateinamen auswählen muß (*TRUE*).

Das folgende Programmlisting enthält die *CanClose*-Elementfunktion und die Elementfunktionen für das Speichern und Laden von Dateien. Das tatsächliche Speichern bzw. das Öffnen der Dateien wird von den Elementfunktionen *SaveFile* und *OpenFile* übernommen.

```
BOOL TMyWindow::CanClose()
{
  if ( !IsDirty )
    return TRUE;
  return MessageBox(HWindow, "Do you want to save?",
    "Drawing has changed", MB_YESNO | MB_ICONQUESTION) == IDNO;
}
```

```
void TMyWindow::CMFileNew(RTMessage)
{
  Points->DeleteAll();
  InvalidateRect(HWindow, NULL, TRUE);
  IsDirty = FALSE;
  IsNewFile = TRUE;
}

void TMyWindow::CMFileOpen(RTMessage)
{
  if ( GetApplication()->ExecDialog(new TFileDialog(this,
       SD_FILEOPEN, strcpy(FileName, "*.PTS"))) == IDOK )
      OpenFile();
}

void TMyWindow::CMFileSave(RTMessage)
{
  if ( IsNewFile )
    SaveFileAs();
  else SaveFile();
}

void TMyWindow::SaveFileAs()
{
  if ( IsNewFile )
    strcpy(FileName, "");
  if ( GetApplication()->ExecDialog(new TFileDialog(this,
       SD_FILESAVE, strcpy(FileName, "*.PTS"))) == IDOK )
      SaveFile();
}

void TMyWindow::CMFileSaveAs(RTMessage)
{
  SaveFileAs();
}

void TMyWindow::SaveFile()
{
   ofpstream os(FileName);
   os << Points;
   os.close();
   IsNewFile = IsDirty = FALSE;
}

void TMyWindow::OpenFile()
{
   ifpstream is(FileName);
   if ( is.bad() )
     MessageBox(HWindow, "Unable to open file", "File Error",
                MB_OK | MB_ICONEXCLAMATION);
   else
```

```
    {
      Points->DeleteAll();
      is >> Points;
      is.close();
      IsNewFile = IsDirty = FALSE;
      InvalidateRect(HWindow, NULL, 1);
    }
}
```

Hinweis: Diese Programmfragmente finden Sie in STEP9.CPP.

Dateien speichern und laden

Jetzt steht das Gerüst zum Öffnen und Speichern von .PTS-Dateien. Nun müssen aber die tatsächlichen Lade- und Schreib-Aktionen aus dem bzw. in das *Points*-Array durchgeführt werden. Dazu definieren wir am einfachsten *Points* als Stream-Array.

Im ersten Schritt wird *TPointArray* von *Array* und *TStreamable* abgeleitet:

```
_CLASSDEF(TPointArray)
class TPointArray : public Array, public TStreamable
{
public:
  TPointArray(int upper, int lower = 0, sizeType aDelta = 0)
    : Array(upper, lower, aDelta){};
.
.
.
protected:
  virtual void write( Ropstream );
  virtual Pvoid read( Ripstream );
.
.
.
};
```

Hinweis: Dieses Programmfragment finden Sie in STEP9.CPP.

Die Elementfunktionen *write* und *read* von *TPointArray* werden aufgerufen, wenn ein *TPointArray* geschrieben bzw. geladen werden soll. Sie sind nachfogend aufgeführt.

```
Pvoid TMyWindow::read(Ripstream is)
{
  TWindow::read(is);
  is >> Points;
  return this;
}
```

```
void TMyWindow::write(Ropstream os)
{
  TWindow::write(os);
  os << Points;
}
```

Bevor *TPointArray* tatsächlich als stream-fähig bezeichnet werden kann, müssen noch weitere Elementfunktionen definiert werden. Die verbleibenden Funktionen – alles »Einzeiler« – finden Sie in STEP9.CPP im Unterverzeichnis EXAMPLES\STEPS. Eine nähere Erläuterung dieser Elementfunktionen finden Sie in Kapitel 15 »Streams und Objekte«. Lassen Sie diese Elementfunktionen für den Moment außer Acht. Wenn Sie sich die Elementfunktionen *read* und *write* von *TMyWindow* ansehen, stellen Sie fest, daß *TMyWindow* auf ganz einfache Weise stream-fähig genannt werden kann, sobald *TPointArray* es ist.

6. Fenster anzeigen

Zum Schluß wollen wir noch ein Hilfesystem für unsere ObjectWindowsAnwendung implementieren. Sie können sich das Hilfesystem als ein Programm im Programm vorstellen, das jederzeit aktiviert werden kann. Windows-Programme haben hier aufgrund ihrer fensterorientierten Umgebung den Vorteil, daß viele Aktivitäten zusammengefaßt und vereinheitlicht werden können. Anhand der Erstellung des Hilfesystems für die Steps-Anwendung soll Ihnen vor Augen geführt werden, wie man ObjectWindows-Programme in einem modularen Format entwickelt und dadurch die Programmpflege erleichtert.

Schritt 10: Ein Hilfefenster anzeigen

In Schritt 8 haben wir dem Hauptfenster der Steps-Anwendung ein Hilfemenü hinzugefügt. Die Auswahl dieses Menüs hat jedoch nur die Anzeige eines leeren Fensters zur Folge, das eine Instanz von *TWindow* ist. Wir werden nun dieses leere Fenster durch ein sinnvolles Hilfefenster, *THelpWindow*, ersetzen und dabei einige Windows-Steuerelemente von ObjectWindows verwenden: Listen, Bildlaufleisten, Editierfelder, statische Steuerelemente, Schalter und Kombinationsfenster. Das Hilfesystem wird in einem separaten Modul definiert, so daß nur geringe Änderungen im momentanen Quelltext erforderlich sind.

Abbildung 6.1 zeigt das fertige Hilfesystem, das für jedes Steuerelement von ObjectWindows eine kurze Beschreibung liefert.

Module mit ObjectWindows verwenden

Ein Programm-Modul bietet den idealen Rahmen zur Definition von Objekt-Elementfunktionen. Dies ist vor allem bei der Programmierung von Windows-Anwendungen sinnvoll, da hier oft dieselben Fenster (Fensterobjekte in ObjectWindows) von verschiedenen Anwendungen benutzt werden. Wir erstellen unser Hilfesystem als Klasse mit dem Namen *THelpWindow* und speichern ihre Definition in einer Datei mit dem Namen HELPWIND.CPP.

Das Hauptprogramm ändern

Es sind nur zwei Änderungen an der Steps-Anwendung nötig:

- Hinzufügen des Moduls HELPWIND.CPP in die Projekt- oder Makedatei der Anwendung.
- Entfernen der Programmfragmente, die Attribute von HELPWIND.CPP aus *TMyWindow::CMHelp* erzeugen. Dieser Programmtext ist nun im *THelpWindow*-Konstruktor enthalten.

Die Steps-Anwendung reagiert jetzt auf die Auswahl des Menüpunktes *Help* durch die Anzeige eines Hilfefensters (*THelpWindow*). Wir müssen daher noch die *THelpWindow*-Klasse definieren. Dies erfolgt im Modul HELPWIND.CPP.

Ein Modul erzeugen

Das Modul *HelpWind* dient ausschließlich der Definition der Klasse *THelpWindow*. Nach seiner Erstellung kann dieses Modul von jedem anderen ObjectWindows-Programm genutzt werden. Es ist im Grunde nichts anderes als ein Fenster in einem Modul. Das Format des Moduls sieht folgendermaßen aus (die Lücken werden wir später noch füllen):

Der Quelltext aus HELPWIND.H:

```
#ifndef __HELPWIND_H
#define __HELPWIND_H

#ifndef __OWL_H
#include <owl.h>
#endif

#ifndef __LISTBOX_H
```

```
#include <listbox.h>
#endif

#ifndef __EDIT_H
#include <edit.h>
#endif

#define ID_LISTBOX  101
#define ID_BUTTON1  102
#define ID_BUTTON2  103
#define ID_EDIT     104

_CLASSDEF(THelpWindow)
class _EXPORT THelpWindow : public TWindow
{
public:
  PTListBox ListBox;
  PTEdit Edit;
  THelpWindow(PTWindowsObject AParent);
  virtual void SetupWindow();
  virtual void HandleListBoxMsg(RTMessage Msg) =
    [ID_FIRST + ID_LISTBOX];
  virtual void HandleButton1Msg(RTMessage Msg) =
    [ID_FIRST + ID_BUTTON1];
  virtual void HandleButton2Msg(RTMessage Msg) =
    [ID_FIRST + ID_BUTTON2];
  virtual void FillEdit(Pchar SelString);
};

#endif
```

Der Quelltext aus HELPWIND.CPP:

```
#include <stdlib.h>
#include <stdio.h>
#include <dos.h>
#include <string.h>
#include <owl.h>
#include <static.h>
#include <edit.h>
#include <listbox.h>
#include <button.h>
#include "helpwind.h"

THelpWindow::THelpWindow(PTWindowsObject AParent)
  : TWindow(AParent, "Help System")
{
.
.
.
}
```

```
void THelpWindow::SetupWindow()
{
.
.
.
}

void THelpWindow::HandleListBoxMsg(RTMessage Msg)
{
.
.
.
}

void THelpWindow::HandleButton1Msg(RTMessage)
{
.
.
.
}

void THelpWindow::HandleButton2Msg(RTMessage)
{
.
.
.
}

void THelpWindow::FillEdit(Pchar SelString)
{
.
.
.
}
```

Steuerelemente in einem Fenster

Aus Schritt 8 wissen Sie, daß es zwei Arten von untergeordneten Fenstern gibt: selbständige und unselbständige. In dem betreffenden Abschnitt wurden zwei Typen von selbständigen untergeordneten Fenstern zum Hauptfenster hinzugefügt: Dialogfenster und Pop-up-Fenster.

Jetzt wollen wir ein unselbständiges untergeordnetes Fenster hinzufügen, ein *Steuerelement*. Die Steuerelemente sind dem Hilfefenster direkt untergeordnet. Unser Hilfefenster ist ein selbständiges, dem Hauptfenster der Anwendung untergeordnetes Fenster. Es ist daher gleichzeitig ein übergeordnetes und ein untergeordnetes Fenster.

Dieser Status ist zulässig und häufig auch erforderlich.

Was sind Steuerelemente?

Steuerelemente sind sichtbare Einheiten, aus denen sich die Anwenderschnittstelle des Fensters (oder ein Teil davon) zusammensetzt. Beispiele für Steuerelemente sind die Schalter *OK* und *Cancel* in Dialogfenstern oder die Listen zur Anzeige von Dateien. Das Hauptfenster der Steps-Anwendung enthält keine Steuerelemente, wir wollen aber einige in unser Hilfefenster aufnehmen.

Es gibt verschiedene Arten von Steuerelementen wie Listen, Bildlaufleisten, Editierfelder, statische Steuerelemente, Schalter und Kombinationsfenster. Unser Hilfefenster (siehe Abbildung 6.1) enthält eine Liste (das große Fenster in der oberen Hälfte), ein Editierfeld (das große Fenster in der unteren Hälfte), ein statisches Steuerelement (der Text »Help Information:«) und zwei Schalter (*Help* und *Cancel*). Bildlaufleisten sind im Editierfeld und in der Liste enthalten, es handelt sich aber dabei nicht um Bildlaufobjekte. Sie werden meist als optionaler Bestandteil von anderen Steuerelementen und Fenstern eingesetzt, dagegen nur selten als selbständiges Steuerelement verwendet.

Die ObjectWindows-Klasse *TControl* definiert die allgemeinen Verhaltensweisen und Eigenschaften aller Steuerelemente. Die davon abgeleiteten Klassen definieren ausschließlich den Fenster-Steuerelementtyp, dessen Instanzen sie repräsentieren. Zum Beispiel definieren *TEdit und TListBox* die Verhaltensweisen von Editierfenstern bzw. Listen. *TControl* ist von *TWindow* abgeleitet. Im Grunde sind Steuerelemente nichts anderes als »spezialisierte« Fenster.

Steuerelemente für Fenster erstellen

Steuerelemente in Dialogfenstern (zum Beispiel im Datei-Dialogfenster) sind zwar in ihrer Verhaltensweise identisch mit Steuerelementen in Fenstern (zum Beispiel in unserem Hilfefenster), bei ihrer Programmierung gibt es aber einen wichtigen Unterschied. Steuerelemente eines Dialogfensters werden in der Dialogressource definiert. Windows erstellt die Steuerelemente für ein Dialogfenster mit Hilfe der Ressourcendefinition. In diesem Fall sind die Steuerelemente nicht notwendigerweise mit ObjectWindows-Objekten verknüpft. Es kann aber unter Umständen erforderlich sein, daß eine derartige Verbindung hergestellt wird. Wenn die Steuerelemente eines Dialogfensters nicht mit ObjectWindows-Objekten verknüpft sind, müssen sie vom Dialogfenster durch direkte Aufrufe von Windows-Funktionen verwaltet werden.

Hinweis: Sie können ein Steuerelement mit einem ObjectWindows-Objekt verknüpfen. Informationen hierzu finden Sie in Kapitel 11 »Dialogobjekte«.

Die Steuerelemente eines Fensters werden dagegen immer durch ObjectWindows-Objekte repräsentiert. Ein übergeordnetes Fenster verwaltet seine Steuerelemente über die vielen Elementfunktionen, die durch Steuerelementobjekte von ObjectWindows definiert sind. Zum Beispiel wird zur Übernahme des Textes, den ein Anwender aus einer Liste gewählt hat, die Elementfunktion *GetSelString* aufgerufen. Ähnlich wie bei einem Fenster- oder Dialogfensterobjekt existiert auch zu jedem Steuerelementobjekt ein entsprechendes sichtbares Element. Ein Steuerelementobjekt ist mit seinem zugehörigen Steuerelement durch das *HWindow*-Datenelement des Objekts verknüpft. Jedes Steuerelement

hat eine eindeutige ID-Nummer, durch die es von seinem übergeordneten Fenster beim Auftreten eines Steuerelement-Ereignisses (zum Beispiel ein Mausklick) identifiziert werden kann. Um das Programm übersichtlicher zu machen, definieren wir für jede Steuerelement-ID eine Konstante:

```
#define ID_LISTBOX 101   // ID für Liste 1
#define ID_BUTTON1 102   // ID für den ersten Schalter (OK)
#define ID_BUTTON2 103   // ID für den zweiten Schalter (Cancel)
#define ID_EDIT 204      // ID für Editierfeld
```

Steuerelementobjekte als Datenelemente

Manchmal ist es sinnvoll, einen Zeiger auf ein Steurelementobjekt oder auf ein anderes untergeordnetes Fenster als Datenelement in einem Fensterobjekt zu speichern. Dies ist nur für untergeordnete Fenster erforderlich, die später direkt durch einen Aufruf ihrer Objekt-Elementfunktionen manipuliert werden. In unserem Fall trifft das nur für die Liste und das Editierfeld zu. *THelpWindow* speichert jedes dieser Steuerelementobjekte in einem separaten Datenelement.

In dem folgenden Listing sehen Sie einen Teil der Objektdefinition von *THelpWindow*:

```
_CLASSDEF(THelpWindow)

class THelpWindow : public TWindow
{
public:
  PTListBox ListBox;
  PTEdit Edit;
.
.
.
};
```

Nachdem die Steuerelementobjekte erzeugt wurden, können sie mit Elementfunktionsaufrufen geändert werden. Beispielsweise könnte durch den Aufruf von *ListBox->AddString* ein neuer String in die Liste *ListBox* aufgenommen werden. Es wäre zwar auch möglich, über die Liste der untergeordneten Fenster des übergeordneten Fensters auf Steuerelementobjekte von untergeordneten Fenstern zuzugreifen (mit Hilfe der *TWindowsObject*-Elementfunktionen *Next* und *Previous*), die nicht als Datenelemente gespeichert sind; die Verwendung von Datenelementen ist aber bedeutend einfacher.

Steuerelemente verwalten

Jede Fensterklasse, die über Steuerelementobjekte verfügt, muß zur Erstellung dieser Objekte einen Konstruktor definieren. Sie kann aber auch die Elementfunktion *SetupWindow*, die sie von *TWindowsObject* erbt, zur Erstellung von

Steuerelementen überschreiben. *THelpWindow* überschreibt *SetupWindow* zur Initialisierung der Liste und der Auswahl in seiner untergeordneten Liste. Beachten Sie, daß *THelpWindow::SetupWindow* zuerst *TWindows::SetupWindow* und diese Funktion wiederum *TWindowsObject::SetupWindow* aufruft, die dann die Fenster erstellt. Die Fenster werden also als untergeordnete Fenster von *THelpWindow* erstellt.

Das nächste Programmlisting enthält den Konstruktor des Hilfefensters. Der Konstruktor ruft als erstes *DisableAutoCreate* auf, ein Aufruf, der auch für die Konstruktoren Ihrer eigenen Pop-up-Fenster in Frage kommt. Dieser Aufruf kennzeichnet durch das Setzen eines Flags, daß die Erstellung des Pop-up-Fensters nicht an die Erstellung seines übergeordneten Fensters gebunden ist. Derartige Verhaltensweisen sind normalerweise nicht für Pop-up-Fenster geeignet.

Anschließend setzt der *THelpWindow*-Konstruktor die Positions- und Größenattribute. Da *THelpWindow* ohne seine untergeordneten Fenster nicht voll funktionsfähig ist, ist seine Initialisierung erst dann vollständig, wenn seine untergeordneten Fenster erstellt sind. *THelpWindow* übergibt this als ersten Parameter an alle Steuerelement-Konstruktoren. this ist ein Zeiger auf das zu erzeugende *THelpWindow*-Objekt. Der zweite Parameter in allen Steuerelement-Konstruktoraufrufen ist eine Steuerelement-ID. Weitere Parameter geben die Position und die Größe an. Beachten Sie, daß *Listbox* und *Edit* für eine spätere Referenzierung gespeichert werden.

```
THelpWindow::THelpWindow(PWindowsObject AParent) :
   TWindow(AParent, "Help System")
{
  DisableAutoCreate();
  Attr.Style |= WS_POPUPWINDOW | WS_CAPTION;
  Attr.X = 100;
  Attr.Y = 100;
  Attr.W = 300;
  Attr.H = 300;
  ListBox = new TListBox(this, ID_LISTBOX, 20, 20, 180, 80);
  new TButton(this, ID_BUTTON1, "Help", 220, 20, 60, 30, TRUE);
  new TButton(this, ID_BUTTON2, "Cancel", 220, 70, 60, 30, FALSE);
  Edit = new TEdit(this, ID_EDIT, "", 20, 180, 260, 90, 40, TRUE);
  new TStatic(this, -1, "Help Information:", 20, 160, 160, 20, 0);
}
```

Hinweis: Dieses Programmfragment finden Sie in HELPWIND.CPP.

Nach dem Erstellen des Hilfefensters wird *THelpWindow::SetupWindow* aufgerufen, um das Hilfefenster zu initialisieren. Das Hilfefenster wird durch die Erzeugung seiner Steuerelemente und die Initialisierung seiner Listen aufgebaut.

THelpWindow::SetupWindow ruft zuerst *TWindow::SetupWindow* auf, das die untergeordneten Fenster des Hilfefensters erstellt. Danach ruft *THelpWindow::SetupWindow* die Elementfunktionen von *ListBox* auf, um die Liste und die Listenauswahl zu initialisieren. Die Steuerelemente werden automatisch nach der Erstellung angezeigt, da per Voreinstellung *WS_VISIBLE* als Erstellungsstil gesetzt ist.

```
void THelpWindow::SetupWindow()
{
  TWindow::SetupWindow();
  ListBox->AddString("List Boxes");
  ListBox->AddString("Buttons");
  ListBox->AddString("Check Boxes");
  ListBox->AddString("Radio Buttons");
  ListBox->AddString("Group Boxes");
  ListBox->AddString("Scroll Bars");
  ListBox->AddString("Edit Controls");
  ListBox->AddString("Static Controls");
  ListBox->AddString("Combo Boxes");
  ListBox->SetSelIndex(0);
};
```

Hinweis: Dieses Programmfragment finden Sie in HELPWIND.CPP.

Die Aufrufe des Konstruktors von *THelpWindow* und der Elementfunktion *SetupWindow* reichen für die korrekte Anzeige der Steuerelemente im Hilfefenster aus. Die Liste kann geblättert, und die Schalter können betätigt werden, es wird aber keine Aktion ausgeführt. Im nächsten Abschnitt definieren wir eine Antwort auf Steuerelement-Ereignisse.

Steuerelement-Ereignisse beantworten

Wie Sie in Schritt 7 erfahren haben, wird auf Menüereignisse durch die Definition von Elementfunktionen zur Befehlsbeantwortung reagiert. Steuerelement-Ereignisse werden durch die Definition von Elementfunktionen bearbeitet, die auf der ID des untergeordneten Fensters basieren. Zur Identifikation einer Botschaftsantwortfunktion verwenden wir die Summe aus der *ID_FIRST*-Konstante und der ID des Steuerelements:

```
_CLASSDEF(THelpWindow)

class THelpWindow : public TWindow
{
public:
.
.
.
  virtual void HandleListBoxMsg(RTMessage Msg) =
    [ID_FIRST + ID_LISTBOX];
.
.
.
};
```

ID_FIRST ist eine von ObjectWindows definierte Konstante. In unserem Beispiel ist *ID_LISTBOX* der Bezeichner für ein Listen-Steuerelement des über-

geordneten Fensters *TMyWindow*. *HandleListBoxMsg* definiert die Antwort auf alle Meldungen, die von diesem Listen-Steuerelement an *TMyWindow* gesendet werden.

Ein Steuerelement sendet immer dann eine Botschaft an sein übergeordnetes Fenster, wenn ein Ereignis eintritt, von dem das übergeordnete Fenster informiert werden sollte. Diese Ereignisse, die zur Übergabe eines Benachrichtigungscodes des Steuerelements an dessen übergeordnetes Fenster führen, sind vom Typ des Steuerelements abhängig. So senden Listen beispielsweise *LBN_SELCHANGE*-Benachrichtigungscodes, wenn sich ihre Auswahl ändert. Schalter senden *BN_CLICKED*-Botschaften, wenn sie angeklickt werden.

Elementfunktionen zur Beantwortung der Benachrichtigungscodes von Steuerelementen wird ein *Msg*-Parameter vom Typ *RTMessage* übergeben. *Msg.LP.Hi* enthält den Code, der den Typ des eingetretenen Ereignisses bestimmt. *Msg.LP.Lo* enthält das Fenster-Handle des Steuerelements. *Msg.WParam* enthält die ID des Steuerelements.

Tabellen der Codes, die in Benachrichtigungsbotschaften von Steuerelementen enthalten sind, finden Sie im *Windows-Referenzhandbuch*. Geben Sie diesen Botschaftsantwortfunktionen der Deutlichkeit halber Namen wie *HandleListBoxMsg*.

Im folgenden sehen Sie die Definition der *THelpWindow*-Klasse mit den Prototypen der Elementfunktionen:

```
_CLASSDEF(THelpWindow)

class THelpWindow : public TWindow
{
public:
  PTListBox ListBox;
  PTEdit Edit;
  THelpWindow(PTWindowsObject AParent);
  virtual void SetupWindow();
  virtual void HandleListBoxMsg(RTMessage Msg) =
    [ID_FIRST + ID_LISTBOX];
  virtual void HandleButton1Msg(RTMessage Msg) =
    [ID_FIRST + ID_BUTTON1];
  virtual void HandleButton2Msg(RTMessage Msg) =
    [ID_FIRST + ID_BUTTON2];
  virtual void FillEdit(Pchar SelString);
};
```

Hinweis: Dieses Programmfragment finden Sie in HELPWIND.H.

Im nächsten Schritt müssen Sie entscheiden, wie *THelpWindow* auf die Botschaften seiner Steuerelemente antworten soll. Wenn der Anwender den *Cancel*-Schalter (*ID_BUTTON2*) anklickt, soll das Fenster geschlossen werden. Die Implementierung von *HandleButton2Msg* ist daher einfach:

```
void THelpWindow::HandleButton2Msg(RTMessage)
{
  CloseWindow();
}
```

Schalter senden nur zwei Arten von Benachrichtigungsbotschaften, nämlich *BN_CLICKED* und *BN_DOUBLECLICKED*. Hier muß der tatsächlich empfangene Benachrichtigungscode des Steuerelements nicht überprüft werden, weil das Fenster in jedem Fall geschlossen werden soll.

Klickt der Anwender dagegen den Schalter *OK* (*ID_BUTTON1*) an, soll die Steps-Anwendung überprüfen, welcher Punkt in der Liste gewählt wurde, und das Editierfeld (*ID_EDIT*) mit dem entsprechenden Text füllen. *HandleButton1Msg* ruft deshalb *GetSelString* für die Liste (*ID_LISTBOX*) auf, um den Text für den gewählten Listeneintrag zu erhalten. Der Text wird an die Elementfunktion *FillEdit* übergeben. Die Definition von *FillEdit* ist in HELPWIND.CPP enthalten.

```
void THelpWindow::HandleButton1Msg(RTMessage Msg)
{
  char SelString[25];
  ListBox->GetSelString(SelString, sizeof (SelString));
  FillEdit(SelString);
}
```

Zum Abschluß wollen wir noch auf die Botschaft, die durch das doppelte Anklicken eines Listeneintrags erzeugt wird, durch Ausfüllen des Editierfelds mit dem entsprechenden Text antworten. Da Listen eine Vielzahl von Botschaften an ihre übergeordneten Fenster schicken, muß der Botschaftscode vor dem Ausfüllen des Editierfelds überprüft werden. Wenn der Wert *LBN_DBLCLK* lautet, hat der Anwender einen Doppelklick ausgeführt, andernfalls steuert ein Aufruf von *DefWndProc* alle anderen Botschaften, die eine Liste betreffen.

```
void THelpWindow::HandleListBoxMsg(RTMessage Msg)
{
  char SelString[25];

  if ( Msg.LP.Hi == LBN_DBLCLK )
  {
    ListBox->GetSelString(SelString, sizeof (SelString));
    FillEdit(SelString);
  }

  else DefWndProc(Msg);
}
```

Mit den Steuerelementobjekten von ObjectWindows steht Ihnen eine Vielzahl von Elementfunktionen zur Abfrage und Bearbeitung zur Verfügung. Wenn Sie diese Elementfunktionen als Antwort auf Steuerelement-Ereignisse aufrufen, können Sie Ihre Fenster »beleben«, wie dies bei unserem Hilfefenster der Fall ist.

II. ObjectWindows verwenden

7. Überblick über ObjectWindows

ObjectWindows ist eine umfassende Zusammenstellung von Klassen, die die Entwicklung von Programmen, die unter Microsoft-Windows laufen, mit C++ auf effiziente Weise unterstützen.

Im vorliegenden Kapitel finden Sie einen Überblick über die Objekthierarchie. In den restlichen Kapiteln in diesem Teil des Handbuchs werden die verschiedenen Hierarchie-Ebenen detailliert erläutert. Zusätzlich zur Beschreibung der Objekthierarchie erfahren Sie in diesem Kapitel einiges über die grundlegenden Porgrammierprinzipien für die Windows-Umgebung, einschließ lich der Verwendung von Ressourcen, des Aufrufs der Windows-API und des Empfangs und der Bearbeitung von Windows-Botschaften.

ObjectWindows-Konventionen

Die Namen der Klassen in ObjectWindows beginnen mit einem *T*. So erstellt z.B. die Klasse *TDialog* Dialogfensterobjekte. Zu jeder Klassendefinition gehört ein Zeiger auf die Klasse. Der Name des Zeigers beginnt mit einem *P*. So lautet der Name des Zeigers auf *TDialog* beispielsweise *PTDialog*. ObjectWindows definiert für alle T-Klassen folgende Typen:

P*class*, ein Zeiger auf die Klasse *class*. (So ist *PTWindow* beispielsweise als Zeiger auf ein *TWindow* definiert.)

R*class*, eine Referenz auf die Klasse *class*. (So ist *RTWindow* beispielsweise als Referenz auf ein *TWindow* definiert.)

RP*class*, eine Referenz auf einen Zeiger auf die Klasse *class*. (So ist *RPTWindow* beispielsweise als Referenz auf einen Zeiger auf ein *TWindow* definiert.)

PC*class*, ein Zeiger auf die **const**-Klasse *class*. (So ist *PCTWindow* beispielsweise als Zeiger auf ein **const** *TWindow* definiert.)

RC*class*, eine Referenz auf einen Zeiger auf die **const**-Klasse *class*. (So ist *RCTWindow* beispielsweise als Referenz auf ein **const** *TWindow* definiert.)

Hinweis: Das Makro _CLASSDEF aus def.h unterstützt diese Namenskonventionen. So erzeugt z.B. der Aufruf von `_CLASSDEF(TMyWindow)` die Typbezeichner

PTMyWindow, RTMyWindow, RPTMyWindow, PCTMyWindow. In den Klassendeklarationen der STEP-Programme aus den vorangegangenen Kapiteln wird davon reger Gebrauch gemacht.

Die beschriebenen Namenstypen dienen der Vereinfachung komplizierter Windows-Programmierprozeduren.

Botschaftsantwortfunktionen werden nach der zu beantwortenden Botschaft benannt, führen jedoch keinen Unterstrich in ihrem Namen. So würde eine Elementfunktion, die auf die Botschaft *WM_KEYDOWN* antwortet, *WMKeyDown* heißen und eine Antwort auf die Botschaft *CM_FILEOPEN* hätte den Namen *CMFileOpen*.

Die ObjectWindows-Hierarchie

ObjectWindows ist eine Bibliothek, die aus einer Hierarchie von Klassen besteht, die Sie – durch Gebrauch der Vererbung – verwenden, ändern und ergänzen können. Kapitel 22 »ObjectWindows-Klassen« enthält eine vollständige Referenz für diese Klassen und ihre Datenelemente und Elementfunktionen (siehe Abbildung 7.1).

Object

Object ist die Basisklasse für alle abgeleiteten ObjectWindows-Klassen. *Object* ist in der Standard-Container-Klassenbibliothek von ObjectWindows definiert. Verwechseln Sie *Object* nicht mit *TObject* aus der Turbo-Vision-Hierarchie. Die beiden Klassen sind völlig unabhängig von einander.

TModule

Die dynamischen Linkbibliotheken von ObjectWindows (DLLs) erstellen eine Instanz von *TModule*, das als objekt-orientierter Ersatz für das Bibliotheksmodul (DLL) dient. ObjectWindows-Anwendungen erzeugen eine Instanz von *TApplication*, das von *TModule* abgeleitet ist. *TModule* definiert das Verhalten, das dem Bibliotheks- und Anwendungsmodul gemeinsam ist. *TModule*-Elementfunktionen unterstützen die Speicherverwaltung für Fenster und die Fehlerbehandlung.

TApplication

Diese Klasse bestimmt Verhalten und Eigenschaften aller ObjectWindows-Anwendungen. Jede ObjectWindows-Anwendung, die Sie schreiben, leitet ihre eigene Anwendungsklasse von *TApplication* ab. Sie ist unter anderem für die Initialisierung des Hauptfensterobjekts verantwortlich. Anwendungsklassen werden detailliert in Kapitel 8 »Anwendungsobjekte- und Module« erläutert.

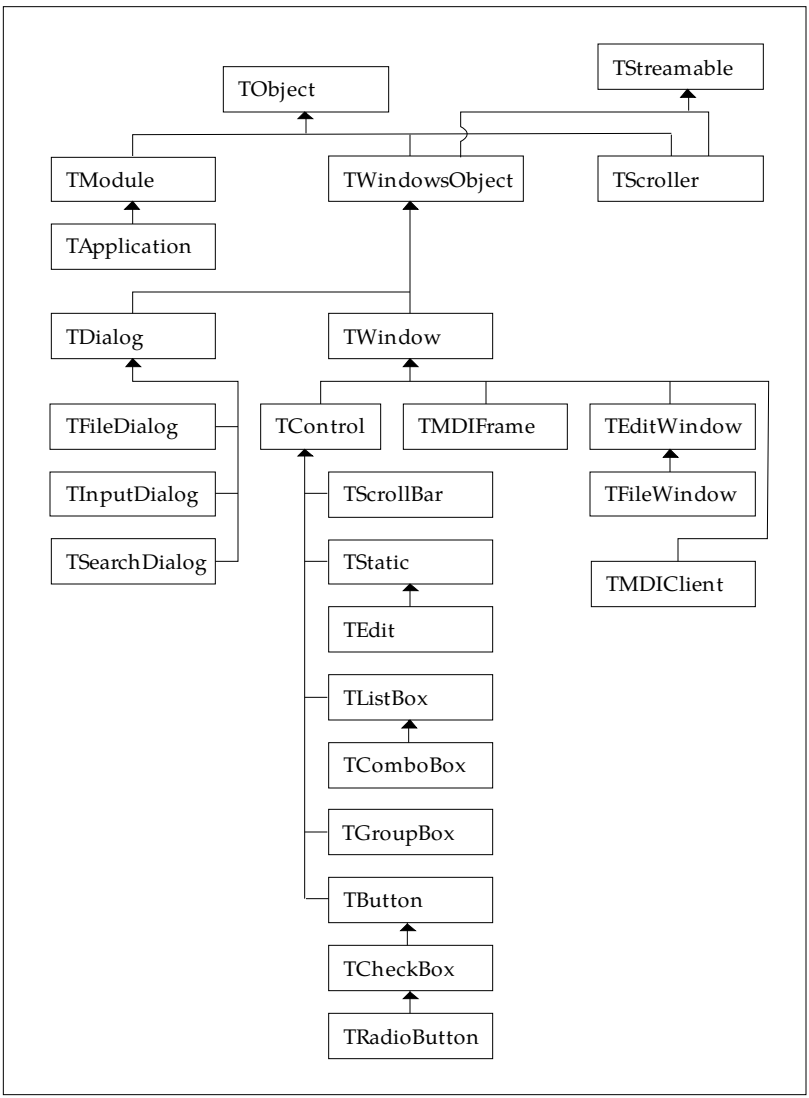


Abbildung 7.1 Die Hierarchie in ObjectWindows

Schnittstellen-objekte

Die restlichen Objekte in der ObjectWindows-Hierarchie werden generell als *Schnittstellenobjekte* klassifiziert. Sie sind dies in zweierlei Hinsicht: Sie repräsentieren Elemente in der Windows-Anwenderschnittstelle und dienen als eine Art

Schnittstelle zwischen Ihrem Programm und der Windows-Umgebung. Schnittstellenobjekte werden in Kapitel 9 »Schnittstellenobjekte« detailliert erläutert.

TWindowsObject
TWindowsObject ist eine abstrakte Basisklasse, die die drei Haupttypen von ObjectWindows-Schnittstellenobjekten in sich vereint: Fenster, Dialogfenster und Steuerelemente. Diese Klasse stellt Elementfunktionen zur Verfügung, die die Erstellung, Botschaftsbearbeitung und die Freigabe von Fensterobjekten verwalten.

Fensterobjekte

Zu den Fensterobjekten zählen nicht nur die schon bekannten Fenster der Windows-Umgebung, sondern auch die meisten der visuellen Elemente innerhalb der Umgebung, wie zum Beispiel Steuerelemente.

TWindow
TWindow ist eine »Allzweck«-Fensterklasse, deren Instanzen als Hauptfenster einer Anwendung oder als eines ihrer Pop-up-Fenster dienen können. Instanzen von *TWindow* können Grafiken anzeigen. Sie werden aber in der Praxis abgeleitete Klassen definieren, um das Verhalten und die Eigenschaften von *TWindow* spezieller zu gestalten.

TEditWindow
Abgeleitet von *TWindow* definiert *TEditWindow* eine Klasse, die das Editieren von Text in einem Fenster ermöglicht.

TFileWindow
Abgeleitet von *TEditWindow* definiert *TFileWindow* eine Klasse, die Textdateien lädt, editiert und speichert.

Dialogobjekte

Dialogobjekte erleichtern den Umgang mit interaktiven Gruppen, vor allem mit Steuerelementgruppen wie Schaltern, Listen und Bildlaufleisten. Dialogobjekte werden in Kapitel 11 »Dialogobjekte« detailliert erläutert.

TDialog
Diese Klasse dient als Basis für abgeleitete Klassen, die Windows-Dialogfenster verwalten. Dialogobjekte sind mit Dialogressourcen verknüpft und können als modale oder nicht-modale Dialogfenster eingesetzt werden. In Kapitel 11 »Dialogobjekte« finden Sie im Abschnitt »Modale und nicht-modale Dialoge« nähere Erläuterungen zu diesem Thema. Außerdem stehen Elementfunktionen

zur Verfügung, die die Kommunikation zwischen Dialogfenstern und ihren Steuerelementen erleichtern.

TFileDialog
TFileDialog ist eine vordefinierte Dialogklasse, die in vielen Anwendungen sehr hilfreich ist. Sie definiert einen Dialog, der dem Anwender die Auswahl einer Datei für verschiedene Zwecke ermöglicht, wie zum Beispiel zum Öffnen, Editieren oder Speichern.

TInputDialog
Diese Klasse definiert ein vordefiniertes Dialogfenster, das dem Anwender die Eingabe einer Textzeile erlaubt.

Steuerelement-objekte

In Dialog- und einigen anderen Fenstern erlauben es die Steuerelemente dem Anwender, Daten einzugeben oder Optionen auszuwählen. Steuerelementobjekte stellen konsistente und einfache Mittel für den Umgang mit den verschiedenen, von Windows definierten Steuerelementen zur Verfügung. Die Objekte werden in Kapitel 12 »Steuerelementobjekte« detailliert erläutert.

TControl
TControl ist eine abstrakte Klasse, die als gemeinsame Vorfahrenklasse für alle Windows-Steuerelementobjekte dient, einschließlich Listen und Schalter. Sie definiert Elementfunktionen, die für ihre abgeleiteten Klassen Steuerelemente anlegen und Botschaften bearbeiten. In Abbildung 12.1 finden Sie Beispiele für einige dieser Steuerelemente.

TButton
Instanzen von *TButton* repräsentieren Windows-Aktionsschalter, wie "OK" und "Abbruch".

TCheckBox
TCheckBox-Instanzen stellen Windows-Auswahlfelder dar und verfügen über Elementfunktionen zur Verwaltung des Status der Felder. *TCheckBox* ist von *TButton* abgeleitet.

TListBox
TListBox-Instanzen ermöglichen die Erstellung von und die Auswahl aus Windows-Listen. *TListBox* definiert Elementfunktionen zur Manipulation von Listeneinträgen.

TComboBox
Abgeleitet von *TListBox* definiert *TComboBox* das Verhalten und die Eigenschaften von Windows-Kombinationsfenstern. Ein Kombinationsfenster ist eine Kombination aus Liste und Editierfeld.

TGroupBox
TGroupBox-Instanzen regeln die Erstellung von Windows-Gruppen.

TRadioButton
Abgeleitet von *TCheckBox* regeln *TRadioButton*-Instanzen die Erstellung und Status-Verwaltung von Windows-Schaltfeldern.

TStatic
TStatic stellt Elementfunktionen zur Verfügung, die den Text eines statischen Steuerelements (nur Ausgabe) setzen, abfragen und löschen.

TEdit
Abgeleitet von *TStatic* unterstützt *TEdit* die leistungsstarken Textbearbeitungsmöglichkeiten für Windows-Editierfelder.

TScrollBar
Diese Klasse definiert Elementfunktionen, die den Bereich und den Positionszeiger einer selbständigen Bildlaufleiste verwalten.

MDI-Objekte

Windows implementiert einen Standard für die Verwaltung von mehreren Dokumenten im Rahmen eines einzelnen Fensters, der MDI (Multidokumentschnittstelle) genannt wird. ObjectWindows stellt Mittel für den Aufbau und die Manipulation von MDI-Fenstern zur Verfügung. MDI-Objekte werden in Kapitel 14 »MDI-Objekte« detailliert erläutert.

TMDIFrame
TMDIFrame sorgt dafür, daß das Verhalten und die Eigenschaften des Hauptfensters einer Anwendung der MDI-Spezifikation von Windows entsprechen.

TMDIClient
TMDIClient unterstützt zusätzlich MDI-Fenster. Das *TMDIClient*-Objekt ist das Objekt, das tatsächlich den MDI-Client-Bereich des Fensters verwaltet, also den Bereich, der die untergeordneten MDI-Fenster (»Dokumente«) aufnimmt.

Scroller-Objekte

TScroller ist das ObjectWindows-Objekt, das die Bildlaufleisten des Fensters bearbeitet und die Möglichkeit zum automatischen Bildlauf von Text und Grafiken im Fenster zur Verfügung stellt. *TScroller* führt auch dann den Bildlauf für die Fenster durch, wenn der Anwender die Maus aus dem Client-Bereich des

Fensters bewegt. *TScroller* arbeitet also auch in Fenstern, die nicht mit Bildlaufleisten versehen sind.

Windows-Funktionen

Die Windows-API umfaßt ca. 600 Funktionen. Windows-Anwendungen verändern die Umgebung, ändern die Anzeige (das Aussehen) oder reagieren als Antwort auf Anwendereingaben durch den Aufruf von Windows-Funktionen. Mit ObjectWindows können Sie jedoch Fenster anlegen, Dialogfenster anzeigen und Steuerelemente manipulieren, ohne eine einzige Windows-Funktion aufrufen zu müssen. Wie ist dies möglich?

ObjectWindows-Aufrufe von Windows-Funktionen

Viele Elementfunktionen von ObjectWindows rufen Windows-Funktionen auf. Dennoch kopiert ObjectWindows nicht deren Funktionalität, es faßt vielmehr einige Windows-Funktionen in einer objektorientierten Bibliothek erneut zusammen. ObjectWindows vereinfacht spürbar die für Windows-Funktionen notwendige Angabe zahlreicher Parameter. Häufig stellt ObjectWindows selbständig Parameter zur Verfügung, wie beispielsweise Fenster-Handle und die ID untergeordneter Fenster. Diese sind als Datenelemente in den Schnittstellenobjekten gespeichert.

So erfordern viele Windows-Funktionen beispielsweise ein Handle, mit dem das betreffende Fenster identifiziert wird. Diese Funktionen werden normalerweise von den Elementfunktionen eines Fensterobjekts aufgerufen. Das Objekt speichert das Handle, das mit seinem Fenster verknüpft ist, im Datenelement *HWindow*. Dieses Datenelement kann als Handle übergeben werden, so daß Sie es nicht jedesmal selber angeben müssen. Das Fenster-Handle und viele andere von Windows benötigte Informationsbits sind in dem Objekt *gekapselt.* Auf diese Weise dienen die ObjectWindows-Klassen als objektorientierte Schnittstelle zu der nicht objektorientierten Windows-API.

Zugriff auf Funktionen

Alle Windows-API-Funktionen stehen einem ObjectWindows-Programm weiter zur Verfügung (siehe Kapitel 1 »Windows-Funktionen« im *Windows-Referenzhandbuch* oder die Datei windows.h. An diesen Stellen finden Sie eine Zusammenstellung der verfügbaren Funktionen).

Der folgende Programmtext ruft z.B. die Windows-Funktion *MessageBox* zur Erstellung eines Meldungsfensters auf:

```
Reply = MessageBox(HWindow, "Do you want to save?",
```

```
"File has changed", MB_YESNO |
MB_ICONQUESTION);
```

Viele Windows-Funktionen liefern wichtige Werte zurück. Bei *MessageBox* ist der Rückgabewert eine Integerzahl, die in unserem Beispiel in *Reply* gespeichert wird. Der Wert der Integerzahl gibt die Aktion an, die der Anwender zum Schließen des Meldungsfensters ausgeführt hat. Wenn der Anwender den *Ja*-Schalter angeklickt hat, entspricht der Wert der von Windows definierten Konstante *IDYES*. Hat der Anwender den *Nein*-Schalter angeklickt, ist das Resultat *IDNO*.

Windows-Funktionen benötigen eine Vielzahl von *WORD*- und *DWORD*-Konstanten als Argumente. Diese Konstanten repräsentieren Fenster-, Dialogfenster- oder Steuerelementstile, Rückgabewerte und vieles mehr. Die Verwendung dieser Konstanten trägt dazu bei, daß Ihr Programm lesbarer wird, leichter zu verwalten ist und unabhängig von Änderungen zukünftiger Windows-Versionen wird. So ist es beispielsweise weitaus informativer, ein Meldungsfenster mit dem Typ *MB_YESNO | MB_ICONQUESTION*, statt mit 0x0024 zu definieren. *MB_YESNO*, *IDYES* und weitere Konstanten sind in windows.h definiert.

In windows.h sehen Sie, daß die Funktion *MessageBox* folgendermaßen deklariert ist:

```
int FAR PASCAL MessageBox(HWND, LPSTR, LPSTR, WORD);
```

Das Wort »int« sollte Ihnen bekannt sein. Was aber bedeuten *FAR*, *PASCAL*, *HWND*, *LPSTR* und *WORD*? Es handelt sich dabei um neue, in windows.h per #define-Direktive erzeugte »Typen«. *FAR* und *PASCAL* sind identisch mit **far** und **pascal**. In windows.h sind auch für **near**, **void** und **long** die groß geschriebenen Entsprechungen definiert. Alle Windows-API-Funktionen sind Funktionen vom Typ *FAR PASCAL*. Die einzigen Ausnahmen bilden *wsprintf* und *wvsprintf*, die vom Typ *FAR* cdecl sind.

Einige Funktionen erfordern komplexere Datenstrukturen, wie beispielsweise diejenigen, die Schriften (LOGFONT) oder Fensterklassen (WNDCLASS) bearbeiten. Normalerweise übergeben Sie Zeiger auf diese Strukturen als Parameter in den Windows-Funktionen, die diese Zeiger benötigen. Eine Zusammenstellung der verfügbaren Strukturen finden Sie im *Windows-Referenzhandbuch*.

Sie müssen windows.h nicht in Ihre Anwendung einbinden. Dies wird automatisch von owl.h durchgeführt.

Stilkonstanten kombinieren

Windows-Funktionen, die Schnittstellenelemente produzieren, benötigen normalerweise Stilparameter vom Typ *WORD* oder *DWORD*. Windows definiert hunderte von Stilkonstanten, die Sie in Kapitel 4 »Windows-Stile und -Konstanten« im *Windows-Referenzhandbuch* finden. Bezeichner für Stilkonstanten bestehen aus einem zweibuchstabigen Präfix, gefolgt von einem Unterstrich und einem beschreibenden Namen. So ist *WS_POPUP* zum Beispiel eine Fensterstilkonstante für Pop-up-Fenster, wobei *WS_* »window style« bedeutet.

Diese Stile müssen zur Erzeugung eines anderen Stils häufig mit Hilfe des bitweisen ODER-Operator (|) kombiniert werden. Im *MessageBox*-Beispiel übergeben Sie *MB_YESNO | MB_ICONQUESTION* als Stilparameter. Dieser Stil erzeugt ein Meldungsfenster mit den beiden Schaltern *Ja* und *Nein* und einem Fragezeichensymbol.

Denken Sie aber daran, daß sich einige Stile gegenseitig ausschließen. Durch die Kombination unverträglicher Stile kann es zu unvorhersehbaren und unerwünschten Ergebnissen kommen.

Typen von Windows-Funktionen

Im folgenden finden Sie die Arten von Windows-Funktionen, die Sie für Ihre ObjectWindows-Programme nutzen sollten.

Schnittstellenfunktionen der Fensterverwaltung

Diese Funktionen bearbeiten Botschaften, manipulieren Fenster und Dialogfenster und erstellen Systemausgaben. Die Kategorie umfaßt Funktionen für Menüs, Mauszeiger und die Zwischenablage.

Funktionen der grafischen Geräteschnittstelle (GDI)

Diese Funktionen geben Text, Grafiken und Bitmaps auf einer Vielzahl verschiedener Geräte, einschließlich Bildschirm und Drucker, aus. Die Funktionen sind nicht an ein bestimmtes Gerät gebunden, sie sind *geräteunabhängig*.

Schnittstellenfunktionen der Systemdienste

Diese Funktionen handhaben eine Vielzahl von Systemdiensten, einschließlich der Speicherverwaltung, der Schnittstelle zum Betriebssystem, der Ressourcenverwaltung und der Kommunikationsvorgänge.

Callback-Funktionen

Callback-Funktionen sind vom Anwender bereitgestellte Funktionen, die von programmexternen Routinen aufgerufen werden. So ruft Windows eine Ihrer Callback-Funktionen auf, wenn innerhalb von Windows vorhandene Objekte aufgelistet werden sollen (wie beispielsweise Schriften oder Fenster). Wenn Sie eine derartige Anforderung an Windows stellen, müssen Sie Windows die Adresse der Callback-Funktion mitteilen.

Wird eine Callback-Funktion aufgerufen, erfolgt dieser Aufruf – wie bereits erwähnt – in einem anderen als Ihrem Programm-Kontext – normalerweise innerhalb von Windows. Damit die Callback-Funktion auf die in Ihrem Programm definierten globalen Variablen (die im Datensegment Ihres Programms enthalten sind) zugreifen kann, müssen Sie sicherstellen, daß das aktuelle Datensegment dem des Programms entspricht. Dazu werden innerhalb von Windows die sog. »Instanzen-Thunks« verwendet. An Windows wird die Adresse eines Instanzen-Thunks übergeben und nicht die tatsächliche Adresse der

Callback-Funktion. Ein Instanzen-Thunk setzt das Datensegment-Register so, daß es auf Ihr Datensegment zeigt und übergibt dann die Steuerung an die Callback-Funktion. Instanzen-Thunks für Callback-Funktionen werden mit der Windows-Funktion *MakeProcInstance* erstellt. Sehen Sie sich dazu das folgende Beispiel an:

```
FARPROC ThunkPtr = MakeProcInstance((FARPROC) MyCallback,
                                    GetApplication()->hInstance);
```

Das erste Argument für *MakeProcInstance* ist ein Zeiger auf Ihre Callback-Funktion, das zweite ist das Instanzen-Handle Ihrer Anwendung.

Alle Callback-Funktionen müssen mit dem Schlüsselwort **_export** exportiert werden. So könnten Sie eine Callback-Funktion folgendermaß en deklarieren:

```
BOOL FAR PASCAL _export MyCallBack(HWND hWnd, DWORD lParam);
```

Alle Callback-Funktionen setzen die Pascal-Aufrufsequenz voraus und werden deshalb mit dem Schlüsselwort **PASCAL** deklariert.

Da Callback-Funktionen von auß erhalb der ObjectWindows-Umgebung aufgerufen und nicht von einer C++-Umgebung aktiviert werden, können keine nicht-statischen C++-Elementfunktionen als Callback-Funktionen verwendet werden. Nicht-statische Elementfunktionen benutzen einen verborgenen Parameter (*this*), den Windows aber nicht unterstützt. Statische Elementfunktionen verwenden diesen *this*-Zeiger nicht und sind deshalb als Callback-Funktionen einsetzbar. Stellen Sie sicher, daß das Schlüsselwort **PASCAL** bei der Deklaration dieser statischen Callback-Funktionen angegeben wird, wenn Windows sie aufrufen soll.

Aufzählungsfunktionen

Die am häufigsten verwendeten Windows-Callback-Funktionen sind diejenigen, die von Windows-Aufzählungsfunktionen aufgerufen werden. Mit Hilfe dieser Funktionen können Sie bestimmte Elementtypen im Windows- System einzeln abrufen. So könnten Sie beispielsweise die Schriften im System auflisten und für jede Schrift ein Textbeispiel drucken. Zur Verwendung dieser Funktio - nen müssen Sie Windows (wie im Vorausgehenden erläutert) die Adresse Ihrer Callback-Funktion mitteilen. Die folgenden Windows-Funktionen erfordern Callback-Funktionen: *EnumChildWindows, EnumClipboardFormats, EnumFonts, EnumMetaFile, EnumObjects, EnumProps, EnumTaskWindows* und *EnumWindows.*

So könnten Sie beispielsweise eine statische C++ Elementfunktion mit dem Namen *TMyWindow::ActOnWindow* folgendermaß en deklarieren:

```
class TMyWindow
{
.
.
.
  static BOOL FAR PASCAL _export TMyWindow::ActOnWindow
     (HWND hWnd, DWORD lParam);
.
.
```

```
.
};
```

Danach könnten Sie diese Funktion als Callback-Funktion in einem Aufruf der Windows-Funktion *EnumWindows* übergeben:

```
FARPROC lpEnumWinFunc;
.
.
.
lpEnumWinFunc = MakeProcInstance((FARPROC) TMyWindow::ActOnWindow,
                                 GetApplication()->hInstance);
ReturnVal = EnumWindows(lpEnumWinFunc, aLongVal);
FreeProcInstance(lpEnumWinFunc);
```

Callbacks für Aufzählungsfunktionen müssen einen Wert ungleich Null zurückliefern, damit die Aufzählung fortgesetzt wird. Im obigen Beispiel würde die statische Elementfunktion *TMyWindow::ActOnWindow* Aktionen in dem vom übergebenen Handle bestimmten Fenster ausführen. Der Parameter *aLongVal* ist ein beliebiger Wert, den der Aufrufer von *EnumWindows* zur Übergabe an die Callback-Funktion auswählt.

»smart callbacks« verwenden

Sie können die zusätzlichen Arbeitsschritte für den Aufruf von *MakeProcInstance* und *FreeProcInstance* umgehen, indem Sie Ihren Programmtext mit der »smart callbacks«-Option compilieren. Diese Option ändert den Prolog- und Epilogcode, so daß DS == SS unterstellt wird. Anders ausgedrückt: Das voreingestellte Datensegment ist dasselbe wie das Stacksegment, wodurch die Aufrufe von *MakeProcInstance* und *FreeProcInstance* entfallen. Das vorausgehende Beispiel würde bei Compilieren mit »smart callbacks« folgendermaßen aussehen:

```
ReturnVal = EnumWindow((FARPROC) TMyWindow::ActOnWindow, aLongVal);
```

Windows-Botschaften

Wenn Windows ein Ereignis registriert, das eine Anwendung betrifft, sendet es eine Botschaft an die Anwendung.

Es gibt verschiedene Botschaftstypen, die von folgenden Vorgängen produziert werden:

- Der Anwender erzeugt durch Bewegen oder Betätigen der Maus oder durch Eingaben über die Tastatur eine Anwendereregnis-Botschaft.
- Ihr Programm kann Botschaften direkt an sich selbst senden.
- Ihr Programm kann Windows-Funktionen aufrufen, die zum Empfang von Windows-Botschaften führen.

- Anwendungen können Botschaften durch dynamischen Datenaustausch (DDE) an andere Anwendungen senden.
- Windows selbst kann Botschaften an Ihre Anwendung senden.

Eine Windows-Anwendung ist für das Ermitteln und Beantworten der für sie bestimmten Botschaften verantwortlich. Dazu kann die Anwendung eine Funktion aufrufen, die für die Beantwortung eines ein bestimmtes Fenster betreffenden Ereignisses definiert wurde.

ObjectWindows führt das Ermitteln und Nachverfolgen der Botschaften für Sie durch. Außerdem unterstützt ObjectWindows den Standard-Bearbeitungsprozeß für eingehende Botschaften. Sie müssen lediglich diejenigen Antworten definieren, die für Ihre Anwendung spezifisch sind. Dazu deklarieren und definieren Sie spezielle *Botschaftsantwortfunktionen* für die Klassen, die Sie aus ObjectWindows-Klassen abgeleitet haben.

Im folgenden sehen Sie eine Elementfunktion-Deklaration, die ein Klicken der linken Maustaste beantwortet, wie sie in der Definition einer von *TWindow* abgeleiteten *TMyWindow*-Klasse erscheinen könnte:

```
class TMyWindow : public TWindow {
public:
.
.
.
  virtual void WMLButtonDown(RTMessage Msg) =
    [WM_FIRST + WM_LBUTTONDOWN];
.
.
.
};
```

Die obige Form der Deklaration gilt für alle Botschaftsantwortfunktionen. C++-Deklarationen von Elementfunktionen wurden dahingehend erweitert, daß ein Integer-Bezeichner mit einer virtuellen Elementfunktion verknüpft werden kann. Dadurch kann ObjectWindows die Botschaften für Ihre Anwendung an deren Botschaftsantwortfunktionen weitergeben.

Wenn Sie wollen, daß eine Instanz von *TMyWindow* ein akustisches Signal gibt, wenn geklickt wird, definieren Sie dieses Verhalten einfach in einer Botschaftsantwortfunktion:

```
void TMyWindow::WMLButtonDown(RTMessage Msg)
{
  MessageBeep(0); // Windows-Funktionsaufruf
}
```

Die Parameter einer Windows-Botschaft

Empfängt eine Windows-Anwendung eine Meldung, die für sie bestimmt ist, werden vier Parameter übergeben. Zwei Parameter enthalten das Handle (den

Windows-Bezeichner) des betreffenden Fensters und den Botschaftsbezeichner. Ihre Anwendung benötigt diese beiden Parameter nicht, da ObjectWindows die Weiterleitung der eingehenden Botschaft an die Botschaftsantwortfunktion eines Fensterobjekts durchführt.

Die beiden anderen Parameter, Werte vom Typ *WORD* und *LONG*, enthalten Daten, die für das Programm nützlich sind. Diese Parameter sind als *WParam*- und *LParam*-Elemente in einer von ObjectWindows definierten *TMessage*-Struktur gepackt. Eine Referenz auf diese Struktur vom Typ *RTMessage* wird an Ihre Botschaftsantwortfunktionen übergeben. Beachten Sie aber, daß die Struktur *TMessage* ein intern von ObjectWindows gesetztes *Result*-Element enthält, das für Sie normalerweise nicht von Interesse ist.

WParam ist vom Typ *WORD* und enthält Daten wie Handle, Bezeichner oder Code. Beispielsweise erzeugt eine Menüauswahl eine *WM_COMMAND*-Botschaft, deren Parameter *wParam* der ID des ausgewählten Menüpunktes entspricht.

LParam kann einen Wert vom Typ *LONG* enthalten, wie beispielsweise einen **far**-Zeiger, oder es beinhaltet zwei Arten von Informationen, eine im niederwertigen und eine im höherwertigen *WORD*. So enthält z.B. *LParam*, das an eine Botschaftsantwortfunktion für *WM_LBUTTONDOWN*-Botschaften (»linke Maustaste gedrückt«) übergeben wird, die x- und y-Koordinaten der Position, an der mit der Maus geklickt wurde.

Um bequemer auf die höher- und niederwertigen *WORD* zugreifen zu können, ist *LParam* in einer Variante zusammen mit einer *LP*-Struktur deklariert, die *Lo*- und *Hi*-*WORD*-Elemente enthält. Wenn Sie *Msg* als eine *TMessage*-Struktur deklariert haben, können Sie mit *Msg.LP.Hi* und *Msg.LP.Lo* einen Bezug auf den höher- bzw. niederwertigen Wert von *Msg.LParam* herstellen. Auf ähnliche Weise ist *TMessage::WParam* in einer Variante zusammen mit einer *WP*-Struktur mit *Hi* und *Lo* als Elementen deklariert, die aber vom Typ BYTE und nicht vom Typ *WORD* sind.

Typen von Windows-Botschaften

Im folgenden finden Sie eine Zusammenstellung der Botschaftstypen, die eine Windows-Anwendung empfangen kann. Eine Liste aller Windows-Botschaften finden Sie in Kapitel 2 »Windows-Botschaften« im *Windows-Referenzhandbuch*.

Botschaften der Fensterverwaltung

Diese Botschaften signalisieren, daß der Status eines Fensters geändert wurde. So wird zum Beispiel *WM_CLOSE* gesendet, wenn ein Fenster geschlossen wird oder *WM_PAINT*, wenn ein Teil eines Fensters oder ein gesamtes Fenster erneut angezeigt werden muß.

Initialisierungsbotschaften

Diese Botschaften werden gesendet, wenn ein Programm ein Menü oder ein

Dialogfenster erzeugt. Zu dieser Gruppe gehören *WM_CREATE* und *WM_INITDIALOG*.

Eingabebotschaften
Diese Botschaften werden als das Ergebnis einer Eingabe mit der Maus, über die Tastatur, die Bildlaufleisten oder die Systemuhr empfangen. Eine der häufigsten Eingabebotschaften ist *WM_COMMAND*, die von einer Menü- oder Tastenkürzelauswahl bzw. von einem Steuerelement-Ereignis, wie dem Anklicken eines Schalters, stammt.

Systembotschaften
Diese Botschaften werden als Antwort auf Anwendermanipulationen am Standardzubehör der Anwendung gesendet, wie beispielsweise am Systemmenüsymbol, den Bildlaufleisten oder bei Betätigung der Schalter für Verkleinerung oder Vergrößerung des Fensters. *WM_SYSCOMMAND* wird als Ergebnis einer Auswahl aus dem Systemmenü gesendet. Die meisten Anwendungen fangen keine Systembotschaften ab.

Zwischenablagebotschaften
Diese Botschaften werden gesendet, wenn eine andere Anwendung auf die Zwischenablage zuzugreifen versucht, während Ihre Anwendung die Zwischenablage besitzt oder überwacht.

Systeminformationsbotschaften
Diese Botschaften werden gesendet, wenn ein Attribut auf der WindowsSystemebene geändert wird, wie beispielsweise die Farbe oder die Schrift.

Botschaften der Manipulation von Steuerelementen
Die Anwendung sendet diese Botschaften an ihre Steuerelemente, beispielsweise an Listen und Schalter. Jede Steuerelementklasse versteht ihre spezifischen Botschaften für das Abfragen und Festlegen von Attributen. Steuerelementbotschaften unterscheiden sich von allen anderen Botschaften – mit Ausnahme von MDI-Botschaften – dadurch, daß sie die Anwendung nicht von einem Ereignis *benachrichtigen*, sondern ein Ereignis *bewirken*. Lesen Sie dazu »Botschaften senden« weiter hinten in diesem Kapitel.

Benachrichtigungsbotschaften von Steuerelementen
Diese Botschaften sind *WM_COMMAND*-Botschaften, die ein übergeordnetes Fenster von einem Steuerelement-Ereignis in Kenntnis setzen, zum Beispiel wenn ein Eintrag aus einer Liste ausgewählt oder etwas in ein Editierfeld eingegeben wird. Die Parameter dieser Botschaften bestimmen einen besonderen Benachrichtigungscode, wie beispielsweise *LBN_SELCHANGE* oder *EN_CHANGE*.

Benachrichtigungsbotschaften von Bildlaufleisten
Diese Botschaften sind spezielle Benachrichtigungsbotschaften für Bildlaufleisten. Es gibt zwei Botschaften dieser Art: *WM_HSCROLL* und *WM_VSCROLL*.

Botschaften von Nicht-Client-Bereichen

Diese Botschaften, einschließ lich *WM_NCMOUSEMOVE* und *WM_NCPAINT*, ähneln den Eingabebotschaften, beziehen sich aber speziell auf Ereignisse auß erhalb des Client-Bereichs des Fensters, einschließ lich Menüzeile, Titelzeile und Rahmen. Es gibt zwei Botschaften dieser Art: *WM_NCMOUSEMOVE* und *WM_-NCPAINT*. Normalerweise ist eine Neudefinition der Bearbeitung dieser Botschaften für Ihre Anwendung nicht nötig.

MDI-Botschaften (Botschaften der Multidokument-schnittstelle)

Diese Botschaften werden von einer Anwendung gesendet, die den MDI-Standard zur Manipulation ihrer untergeordneten Fenster verwendet. Die MDI-Botschaften sind *WM_MDIACTIVATE* und *WM_MDIDESTROY*.

Standardbearbeitung von Botschaften

Windows bietet Standardantworten für viele der Botschaften, die es an eine Anwendung sendet. Wenn Ihr Fenster eine eingehende Botschaft ignoriert, aber keine entsprechende Botschaftsantwortfunktion definiert, ruft ObjectWindows automatisch die von Windows zur Verfügung gestellte Standardbearbeitung auf. Die für ein Fenster geeignete Bearbeitung wird von dessen Elementfunktion *DefWindowProc* festgelegt.

Möglicherweise soll jedoch zusätzlich zur definierten Beantwortungsroutine die Windows-Standardantwort ausgeführt werden. Ist das der Fall, rufen Sie die *DefWindowProc*-Elementfunktion Ihres Fensters aus einer Botschaftsantwort-funktion auf.

Informationen hierzu finden Sie in Kapitel 9 »Schnittstellenobjekte« im Abschnitt »Botschaftsbearbeitung durch Standardantworten«.

Botschaften senden

Eine Quelle für Botschaften ist Ihr Programm selbst. Ihr Programm kann durch den Aufruf der Windows-Funktionen *SendMessage* und *PostMessage* Botschaften an sich senden. Dieses Vorgehen ist dann hilfreich, wenn ein Ereignis von Ihrem Programm simuliert werden soll. Weitere Informationen über diese API-Funktionen stehen in der Online-Hilfe oder im *Windows-Referenzhandbuch* zur Verfügung.

SendMessage sollte vor allem zur Manipulation von Steuerelementen, wie Listen und Schaltern, eingesetzt werden. Windows definiert *Botschaften zur Steuerelementmanipulation*, um zum Beispiel Einträge in eine Liste einzufügen (*LB_ADDSTRING*) oder den Status eines Auswahlfeldes zu ändern (*BM_SETCHECK*). Die Steuerelementobjekte von ObjectWindows definieren Elementfunktionen, wie zum Beispiel *TListBox::AddString* und *TCheck-Box::SetCheck*, die viele dieser häufig verwendeten Botschaften an ihre zugehö-

rigen Schnittstellenelemente senden. Wenn Sie eine seltener verwendete Botschaft an die Steuerelemente senden möchten, verwenden Sie *SendMessage*, wobei *HWindow* des Steuerelements als erster Parameter (*Wnd*) übergeben wird.

Die Vorgehensweise für das Senden einer Botschaft an ein Steuerelement in einem Dialogfenster gestaltet sich etwas anders. Mit den Steuerelementen in Ihren *TDialog*-Objekten sind meist keine ObjectWindows-Schnittstellenobjekte verknüpft. Sie können aber mit Hilfe von *TDialog::SendDlgItemMsg* Botschaften an sie senden. Die Kommunikation mit den Steuerelementen in Dialogfenstern läßt sich vereinfachen, wenn Sie diese Elemente mit Hilfe eines alternativen *TWindow*-Konstruktors mit Schnittstellenelementen verknüpfen. Weitere Informationen finden Sie in Kapitel 11 »Dialogobjekte« im Abschnitt »Dialog-Steuerelemente manipulieren«.

Botschafts-bereiche

Da Botschaften durch Werte vom Typ *WORD* definiert werden, kann es 65536 Botschaften geben, die die Nummern von 0 bis 65535 tragen. Windows teilt Botschaften in eindeutige Bereiche für Standard-Windows-Botschaften, Befehlsbotschaften, Benachrichtigungsbotschaften, etc. ein. Um diese Bereiche leichter identifizieren zu können, definiert ObjectWindows Konstanten, die als Offset für die einzelnen Bereiche verwendet werden.

Botschaftsbereiche und die entsprechenden ObjectWindows-Konstanten finden Sie in der Tabelle 7.1

Tabelle 7.1 Bereiche der Windows-Botschaften

Konstante	Wert	Bereich	Bedeutung
WM_FIRST	0x0000	0x0000-0x03FF	Windows-Botschaften
WM_USER	0x0400	0x0400-0x7EFF	Anwenderdefinierte Fensterbotschaften
WM_INTERNAL	0x7F00	0x7F00-0x7FFF	Reserviert für internen Gebrauch
ID_FIRST	0x8000	0x8000-0x8EFF	Botschaften der untergeordneten Fenster-ID
ID_INTERNAL	0x8F00	0x8F00-0x8FFF	Reserviert für internen Gebrauch
NF_FIRST	0x9000	0x9000-0x9EFF	Anwenderdefinierte Benachrichtigungsbotschaften
NF_INTERNAL	0x9F00	0x9F00-0x9FFF	Reserviert für internen Gebrauch
CM_FIRST	0xA000	0xA000-0xFEFF	Anwenderdefinierte Befehlsbotschaften
CM_INTERNAL	0xFF00	0xFF00-0xFFFF	Reserviert für internen Gebrauch

Anwenderdefinierte Botschaften

Windows ermöglicht es Ihnen, eigene Botschaften für den Einsatz in Ihrer Anwendung zu definieren. Anwenderdefinierte Botschaften sind vor allem bei der Definition und Beantwortung neuer Ereignisse sehr hilfreich. So könnten Sie beispielsweise ein Fenster eine anwenderdefinierte Botschaft an alle anderen offenen Fenster der Anwendung senden lassen.

Zu jeder Windows-Botschaft gehört ein Bezeichner vom Typ *WORD*. Für vordefinierte Windows-Botschaften ist in windows.h eine entsprechende Konstante vom Typ *WORD* definiert. Diese Konstante wird dann in der Deklaration einer Botschaftsantwortfunktion eingesetzt.

Im zulässigen Bereich für Botschaftsbezeichner sind Werte für anwenderdefinierte Botschaften reserviert. Der für vordefinierte Windows-Botschaften reservierte Wertebereich liegt zwischen 0 und *WM_USER* -1 (*WM_USER* ist eine Konstante). Der Wertebereich für anwenderdefinierte Botschaften liegt zwischen *WM_USER* und *WM_USER* +31744. Es ist am besten, wenn anwenderdefinierte Werte als Konstanten definiert werden, die mit den Werten *WM_-USER*, *WM_USER* +1, *WM_USER* +2, usw. beginnen.

Sehen Sie sich das folgende Beispiel an:

```
#define WM_MYFIRSTMESSAGE  = WM_USER;
#define WM_MYSECONDMESSAGE = WM_USER + 1;
#define WM_MYTHIRDMESSAGE  = WM_USER + 2;
```

Senden Sie Ihre Botschaften dann mit der Windows-Funktion *SendMessage*:

```
SendMessage(AWindow->HWindow, WM_MYFIRSTMESSAGE, AWord, ALong);
```

Für den Empfang einer anwenderdefinierten Botschaft gehen Sie wie bei Standard-Windows-Botschaften vor, zum Beispiel:

```
class MyWindow : public TWindow
{
.
.
.
  virtual void WMMyFirstMessage(RTMessage Msg) =
    [WM_FIRST + WM_MYFIRSTMESSAGE];
  virtual void WMMySecondMessage(RTMessage Msg) =
    [WM_FIRST + WM_MYSECONDMESSAGE];
  virtual void WMMyThirdMessage(RTMessage Msg) =
    [WM_FIRST + WM_MYTHIRDMESSAGE];
.
.
.
};
```

8. Anwendungsobjekte und -Module

Die wichtigste Anforderung an eine ObjectWindows-Anwendung ist die Definition einer Anwendungsklasse. Diese ist von der ObjectWindows-Klasse *TApplication* abgeleitet. Das Anwendungsobjekt erbt folgende Verhaltensweisen und Eigenschaften einer ObjectWindows-Anwendung:

- Erstellen und Anzeigen des Hauptfensters der Anwendung.
- Initialisieren der ersten Instanz einer Anwendung für Aufgaben, die allen Instanzen der Anwendung dienen, wie zum Beispiel das Konfigurieren einer Schnittstelle.
- Initialisieren aller Instanzen einer Anwendung, beispielsweise durch das Laden einer Tastenkürzeltabelle. (Sie können viele Instanzen derselben ObjectWindows-Anwendung gleichzeitig laufen lassen.)
- Bearbeiten der Windows-Botschaften, die eine Anwendung empfängt.
- Schließen der Anwendung.

Neben der Definition einer Klasse, die von *TApplication* abstammt, müssen Sie die Fähigkeit zur Erstellung des Hauptfensterobjekts hinzufügen. Sie haben auch die Möglichkeit, die voreingestellten Verhaltensweisen für die Initialisierung von Instanzen, das Schließen der Anwendung und das Bearbeiten von Windows-Botschaften zu überschreiben.

Den Anwendungsfluß steuern

Das Hauptprogramm Ihrer ObjectWindows-Anwendung setzt sich normalerweise aus drei Anweisungen zusammen.

Die erste Anweisung

```
THelloApp HelloApp("HelloApp", hInstance, hPrevInstance, lpCmdLine,
                   nCmdShow);
```

erstellt das Anwendungsobjekt durch den Aufruf seines Konstruktors. Der Konstruktor initialisiert auch die Datenelemente des Anwendungsobjekts. Andere Teile eines ObjectWindows-Programms können auf das Anwendungsobjekt zugreifen, indem sie *GetApplication* aufrufen.

Die zweite Anweisung (`HelloApp.Run();`) ruft die Elementfunktion *Run* auf, die dann ihrerseits *InitApplication* und *InitInstance* aufruft. Diese Funktionen führen die Initialisierung der ersten und aller weiteren Instanzen durch. Danach wird zur Erstellung des Hauptfensters *InitMainWindow* aufgerufen. Meistens müssen Sie nur *InitMainWindow* überschreiben (vgl. Abbildung 8.1).

Run startet dann mit dem Aufruf von *MessageLoop* die Anwendung. *MessageLoop* beginnt mit der Bearbeitung der eingehenden Windows-Botschaften, enthält also die Instruktionen, welche die Anwendungsaktivitäten steuern. *MessageLoop* ruft Elementfunktionen auf, die bestimmte eingehende Botschaften beantworten. *MessageLoop* ist genau das, was der Name zum Ausdruck bringt: eine Botschaftschleife, die abläuft, bis die Anwendung geschlossen wird.

Die dritte Anweisung (`return HelloApp.Status;`) liefert den Endstatus der Anwendung, der von ObjectWindows im Datenelement *Status* gespeichert wird. Dies ist erforderlich, weil *WinMain* einen Integerwert zurückgeben muß. Der Status-Wert ist aber auch für das Testen (Debuggen) hilfreich, da ein Wert, der nicht Null ist, einen Fehler signalisiert.

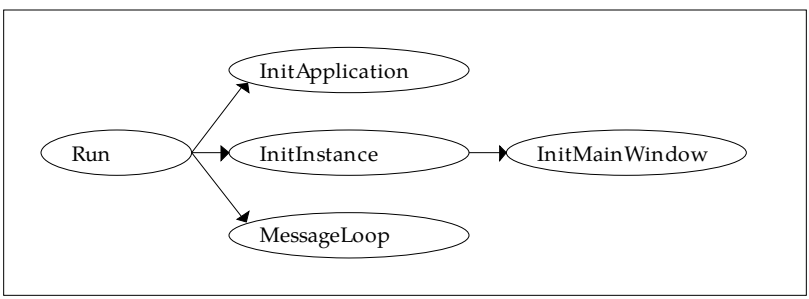


Abbildung 8.1 Elementfunktionsaufrufe, die den Anwendungsfluß steuern

Anwendungen initialisieren

Die Initialisierung des Anwendungsobjekts durch Aufrufe von Elementfunktionen ermöglicht es Ihnen, viele Teile des Prozesses selbst zu gestalten, indem Sie die Elementfunktionen überschreiben. Die einzige Elementfunktion, die Sie schreiben müssen, ist *InitMainWindow*. Sie können auch *InitInstance* zur Initialisierung jeder ausgeführten Instanz einer Anwendung überschreiben, ebenso wie *InitApplication* zur Initialisierung der ersten ausgeführten Instanz einer ObjectWindows-Anwendung.

Das Hauptfensters initialisieren

Im folgenden Listing sehen Sie eine Minimaldefinition für die Anwendungsklasse:

```
class THelloApp : public TApplication
{
public:
  THelloApp(LPSTR AName, HANDLE hInstance, HANDLE
            hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
            : TApplication(AName, hInstance, hPrevInstance,
            lpCmdLine, nCmdShow)  {};
  virtual void InitMainWindow;
};
```

Hinweis: Dieses Programmfragment finden Sie in HELLOAPP.CPP.

Sie müssen eine *InitMainWindow*-Elementfunktion definieren, die das Hauptfensterobjekt erstellt und initialisiert und im Datenelement *MainWindow* des Anwendungsobjekts speichert. Im folgenden sehen Sie ein Beispiel für eine *InitMainWindow*-Elementfunktion:

```
void THelloApp::InitMainWindow()
{
  MainWindow = new TMyWindow(NULL, "Hello, World!");
}
```

Diese einfache ObjectWindows-Anwendung besteht aus einem Hauptprogramm und der Definition der Anwendungsklasse, die von *TApplication* abgeleitet ist und eine Elementfunktion *InitMainWindow* definiert.

Bei dem folgenden Listing handelt es sich um HELLOAPP.CPP.

```
#include <owl.h>

// Define a class derived from TApplication
class THelloApp :public TApplication
{
public:
  THelloApp(LPSTR AName, HANDLE hInstance, HANDLE
            hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
            : TApplication(AName, hIstance, hPrevInstance,
            lpCmdLine, nCmdShow) {};
  virtual void InitMainWindow();
};
// Construct the THelloApp's MainWindow data member
void THelloApp::InitMainWindow()
{
  MainWindow = new TWindow(NULL, "Hello World!");
}
int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow)
{
  THelloApp HelloApp("HelloApp", hInstance, hPrevInstance,
                     lpCmdLine, nCmdShow);
  HelloApp.Run();
  return HelloApp.Status;
}
```

Das obige Programm ist eine Minimalanwendung, die lediglich ein Fenster mit dem Titel "Hello World!" anzeigt (vgl. Abbildung 8.2). Sie können dieses Fenster verschieben und in seiner Größe verändern. Sie können es zu einem Symbol verkleinern, indem Sie das ↓-Symbol in der oberen rechten Ecke anklicken. Wenn Sie den ursprünglichen Zustand wieder herstellen wollen, klicken Sie das Symbol doppelt an. Wenn Sie das Symbol ↑ anklicken, wird das Fenster auf die volle Bildschirmgröße vergrößert. Schließen Sie das Fenster und beenden Sie die Anwendung, indem Sie das Systemmenüsymbol in der oberen linken Ecke doppelt anklicken. Dieses Programm ist eine rudimentäre, aber voll funktionsfähige Anwendung.

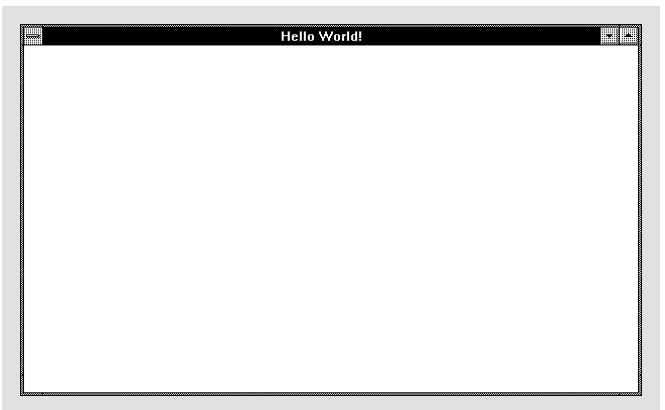


Abbildung 8.2
Das Hauptfenster

Eine Anwendungsinstanz initialisieren

Die Elementfunktion *InitInstance* einer Anwendungsklasse führt die Initialisierung für jede Instanz einer Windows-Anwendung durch. *TApplication* definiert eine *InitInstance*-Elementfunktion, die *InitMainWindow* aufruft und dann das Hauptfenster mit einem Aufruf ihrer *Show*-Elementfunktion anzeigt. Sie müssen nur *InitInstance* überschreiben, um die Standardinitialisierung so zu ändern, daß beispielsweise eine Tastenkürzeltabelle geladen werden kann (eine anwendungsorientierte Aktivität). Wenn Sie *InitInstance* für Ihre Anwendungsklasse überschreiben, müssen Sie sicherstellen, daß *TApplication::InitInstance* zuerst aufgerufen wird.

Im folgenden sehen Sie eine *InitInstance*-Elementfunktion, die eine Tastenkürzeltabelle lädt, bevor die Anwendung gestartet wird. 100 ist der Ressourcenbezeichner der Tastenkürzeltabelle, der in der Ressourcendatei definiert ist:

```
void TMyApp::InitInstance()
{
  TApplication::InitInstance();
  HAccTable = LoadAccelerators(GetApplication()->hInstance,
                               MAKEINTRESOURCE(100));
}
```

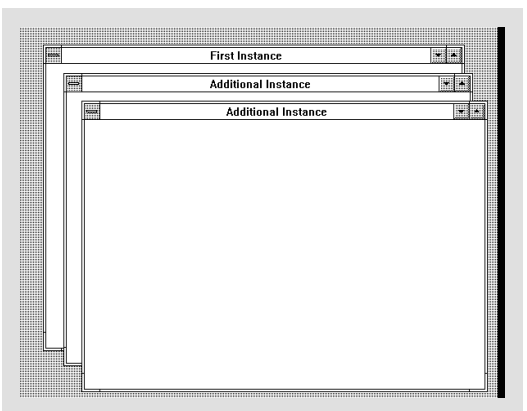
InitInstance sollte nur die Initialisierung der Anwendungsinstanz ausführen. Die Initialisierung des Hauptfensters sollte im Konstruktor sowie in der *SetupWindow*-Elementfunktion des Hauptfensterobjekts erfolgen.

Die erste Anwendungsinstanz initialisieren

Wenn der Anwender Ihre Anwendung mehrmals gleichzeitig ausführt (ohne sie zu beenden), können Sie einen Arbeitsschritt definieren, der nur beim ersten Start durchgeführt wird. Wir nennen dies die Initialisierung der ersten Instanz. Beachten Sie, daß jede Instanz als erste Instanz betrachtet wird, wenn der Anwender die Anwendung startet und beendet, wieder startet, etc.

Wenn die aktuelle Instanz eine erste Instanz ist, wird die Elementfunktion *InitApplication* aufgerufen. *TApplication* definiert eine Dummy-Elementfunktion, *InitApplication*, die zur Ausführung der speziellen Initialisierung der ersten Instanz überschrieben werden kann.

Sie können zum Beispiel *TestApp* so ändern, daß der Titel des Hauptfensters anzeigt, ob es sich um die erste Instanz handelt oder nicht. Zu diesem Zweck fügen Sie ein Datenelement *WindowTitle* in Ihre Anwendungsklasse ein. Definieren Sie dann eine Elementfunktion *InitApplication*, die *WindowTitle* auf "First Instance" setzt. Lassen Sie *WindowTitle* durch den Anwendungskonstruktor auf "Additional Instance" setzen. Dieser Titel gilt dann für alle zusätzlichen Instanzfenster, weil *InitApplication* nur für die erste Instanz aufgerufen wird.



Bildschirm 8.3 Neudefinieren der Anwendungsinitialisierung

Bei dem folgenden Listing handelt es sich um INSTTEST.CPP

```
#include <owl.h>
#include <string.h>

class TTestApp : public TApplication
{
public:
```

```
    TTestApp(LPSTR AName, HANDLE hInstance, HANDLE hPrevInstance,
             LPSTR lpCmdLine, int nCmdShow);

  protected:
    char WindowTitle[20];
    virtual void InitMainWindow();
    virtual void InitApplication();
  };

  TTestApp::TTestApp(LPSTR AName, HANDLE hInstance, HANDLE
    hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
    : TApplication(AName, hInstance, hPrevInstance, lpCmdLine,
      nCmdShow)
  {
    strcpy(WindowTitle, "Additional Instance");
  }

  void TTestApp::InitApplication()
  {
    strcpy(WindowTitle, "First Instance");
  }

  void TTestApp::InitMainWindow()
  {
    MainWindow = new TWindow(NULL, WindowTitle);
  }

  int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
  {
    TTestApp TestApp("Instance Tester", hInstance, hPrevInstance,
      lpCmdLine, nCmdShow);
    TestApp.Run();
    return TestApp.Status;
  }
```

Anwendungen ausführen

Die Botschaftschleife der Anwendung wird gestartet, wenn das Programm seine Anwendungsobjekt-Elementfunktion *Run* aufruft. Diese aktiviert wiederum ihre *MessageLoop*-Elementfunktion. Während der Ausführung des Programms bearbeitet die Botschaftschleife die eingehenden Botschaften. Ihr ObjectWindows-Programm erbt eine solche *MessageLoop*-Elementfunktion.

Die ObjectWindows-*MessageLoop*-Elementfunktion ruft zur Bearbeitung von Windows-Botschaften drei Übersetzungselementfunktionen auf. *ProcessDlgMsg* bearbeitet Botschaften nicht-modaler Dialogfenster, *ProcessAccels* bearbeitet Tastenkürzel und *ProcessMDIAccels* bearbeitet Tastenkürzel für MDI-Anwendungen. Für Anwendungen, die keine Tastenkürzel oder nichtmodale Dialogfenster

verwenden, bzw. die keine MDI-Anwendungen sind, können Sie *MessageLoop* vereinfachen. Näheres dazu finden Sie in der Beschreibung der Übersetzungse - lementfunktionen von *TApplication* in Kapitel 22 »ObjectWindows-Klassen«.

Anwendungen schließen

Bevor Ihre Anwendung beendet werden kann, muß die Botschaftschleife abgebrochen werden. Dies kann erfolgen, wenn der Anwender versucht, das Hauptfenster der Anwendung zu schließen. Wie gesagt, dies*kann* der Fall sein. Denn ObjectWindows verfügt über einen Mechanismus, der das Schließverhalten einer Anwendung neu definiert, beispielsweise vor dem Schließen auf nicht gespeicherte Dateien überprüft.

Im folgenden haben wir die Vorgänge zusammengestellt, die ablaufen, wenn der Anwender durch doppeltes Anklicken des Systemmenüsymbols die Anwendung schließen möchte:

1. Windows sendet eine *WM_CLOSE*-Botschaft an das Hauptfenster der Anwendung.

2. Das Hauptfensterobjekt antwortet darauf mit einem Aufruf der *CloseWindow*-Elementfunktion, die das Hauptfenster erst dann durch Aufruf von *ShutDownWindow* schließt, wenn das Anwendungsobjekt durch einen Aufruf seiner *CanClose*-Elementfunktion überprüft wurde (das Schließen des Anwendungs-Hauptfensters kommt nämlich dem Schließen der Anwendung gleich).

3. Die *CanClose*-Elementfunktion des Anwendungsobjekts liefert das Ergebnis eines Aufrufs der *CanClose*-Elementfunktion des Hauptfensters zurück. Dadurch erlaubt sie dem Hauptfenster selbst zu entscheiden, ob es geschlossen werden soll.

4. Das Hauptfenster entscheidet per Voreinstellung, ob es sich selbst schließen wird, indem es alle untergeordneten Fenster fragt, ob sie das Schließen erlauben. Wenn alle *CanClose*-Elementfunktionen der untergeordneten Fenster *TRUE* zurückliefern, liefert auch *CanClose* des Hauptfensters *TRUE*, so daß die Anwendung beendet werden kann.

Dieser Mechanismus ist vergleichbar mit der Formel »Wenn es jemanden gibt, der weiß, warum die Anwendung nicht geschlossen werden sollte, möge er jetzt sprechen oder für immer schweigen«. Am Ende ist das Anwendungsobjekt für die Erlaubnis zum Schließen der Anwendung verantwortlich. Per Voreinstellung wird das Hauptfenster vor dem Schließen der Anwendung überprüft. Sie können die *CanClose*-Elementfunktion für die Hauptfensterklasse überschreiben, so daß sie ein anderes Schließverhalten zeigt.

9. Schnittstellen-objekte

Objekte zur Darstellung von Fenstern, Dialogfenstern und Steuerelementen werden als Schnittstellenobjekte bezeichnet. Dieses Kapitel befaß t sich mit den Anforderungen und Verhaltensweisen eines Schnittstellenobjekts und mit seiner Beziehung zu den Fenstern, Dialogfenstern und Steuerelementen, die auf dem Bildschirm angezeigt werden. Auß erdem wird das Verhältnis zwischen einzelnen Schnittstellenobjekten einer Anwendung und der Mechanismus für die Übergabe von Windows- Botschaften erläutert.

TWindowsObject

TWindowsObject ist die gemeinsame Basisklasse aller Schnittstellenklassen für Windows: *TDialog*, *TWindow* und die aus *TWindow* abgeleitete Klasse *TControl*. *TWindowsObject* definiert Verhaltensweisen, die allen Fenster-, Dialog- und Steuerelementobjekten gemeinsam sind.

Die Elementfunktionen von *TWindowsObject* erfüllen folgende Aufgaben:

- Verwaltung der dualen Objektstruktur von ObjectWindows und Windows. Dazu gehört auch die Erstellung und das Freigeben und Entfernen von Objekten und Elementen.
- Unterstützung der hierarchischen Beziehungen (Unter- und Überordnung) zwischen den Schnittstellenobjekten (womit nicht deren objektbezogenen Vererbungsbeziehungen gemeint sind).
- Registrierung neuer Fensterklassen (vgl. Kapitel 10 »Fensterobjekte«).

TWindowsObject arbeitet hinter den Kulissen, und Sie werden nur äuß erst selten neue Typen direkt von *TWindowsObject* ableiten. *TWindowsObject* definiert einen groß en Teil der Funktionsweise, die ein Objekt erbt, wenn Sie neue Klassen von *TWindow* und *TDialog* ableiten.

Wofür Schnittstellenobjekte benötigt werden

Wofür werden nun Schnittstellenobjekte benötigt, wenn doch Windows bereits über visuelle Fenster, Dialogfenster und Steuerelemente verfügt?

Zu jedem Schnittstellenobjekt gehört normalerweise ein *visuelles Schnittstellenelement* – nicht ein Objekt, sondern ein physikalisches Fenster, Dialogfenster oder Steuerelement – für das es als objektorientiertes »Äquivalent« agiert. Das Schnittstellenobjekt stellt die Elementfunktionen zur Erstellung, Verwaltung, Initialisierung und zur Freigabe und Entfernung des zugehörigen Schnittstellenelements zur Verfügung. Datenelemente des Schnittstellenelements enthalten die zugehörigen Daten, beispielsweise das entsprechende Handle und die über- und untergeordneten Fenster. Die Elementfunktionen der Schnittstellenobjekte nehmen Ihnen somit einen großen Teil der Arbeit bei der Programmierung für Windows ab.

Man könnte das Verhältnis zwischen Objekt und Element mit demjenigen eines C++-Stream und einer DOS-Datei vergleichen. Sie können ein StreamObjekt erstellen, das die Einzelheiten der Ein-/Ausgabe in eine DOS-Datei verwaltet, ebenso wie Sie ein Fensterobjekt erstellen können, das die Einzelheiten der Ein-/Ausgabe von Fenstern regelt.

Bei der Strukturierung eines Schnittstellenobjekts und dem zugehörigen Schnittstellenelement muß auf die korrekte Behandlung von Fenstern, Dialogfenstern und Steuerelementen geachtet werden. So ist zum Beispiel für die Erstellung eines vollständigen Schnittstellenobjekts der Aufruf von *zwei* Elementfunktionen erforderlich. Die erste besteht aus einem Konstruktor, der das Schnittstellenobjekt erstellt und – falls erforderlich – dessen Attribute setzt (beispielsweise den Stil und das Menü).

Bei der zweiten Funktion handelt es sich um die Erstellungselementfunktion *MakeWindow*, die das Schnittstellenobjekt mit einem neuen Schnittstellenelement verknüpft. Diese Verknüpfung wird durch das *HWindow*-Datenelement des Schnittstellenobjekts verwaltet, in dem ein Handle für ein Fenster gespeichert ist (Handle für Fenster werden in Kapitel 7 »Überblick über ObjectWindows« ausführlich beschrieben). *MakeWindow* ist eine Elementfunktion des Anwendungsobjekts und ruft die *Create*-Elementfunktion des Objekts auf. *Create* weist Windows an, das visuelle Element zu erstellen.

MakeWindow erzeugt auf sichere Art und Weise ein visuelles Element. *MakeWindow* stellt im Unterschied zu *Create* sicher, daß das Schnittstellenobjekt korrekt erstellt wird und daß genügend Speicherplatz zur Verfügung steht. Der Geschwindigkeitsverlust durch den Aufruf von *MakeWindow* gewährleistet umgekehrt, daß genügend Speicherplatz verfügbar bleibt und Windows nicht »abstürzt«.

Ähnlich ist es bei der Zerstörung des Schnittstellenobjekts: Sie müssen das visuelle Schnittstellenelement entfernen und das entsprechende Schnittstellenobjekt freigeben. Hier müssen Sie jedoch nur *ShutDownWindow* aufrufen, das das Schnittstellenelement entfernt und dann das Schnittstellenobjekt freigibt.

Die Erstellung eines Schnittstellenobjekts und des zugehörigen visuellen Schnittstellenelements führt nicht notwendigerweise zu einer Anzeige auf dem Bildschirm. Windows überprüft bei der Erstellung des visuellen Elements, ob für das Fenster der Stil *WS_VISIBLE* gesetzt ist. Das Setzen und Löschen von *WS_VISIBLE* und anderen Windows-Stilen erfolgt im Konstruktor, indem diese Stile im Datenelement *Attr.Style* des Objekts gesetzt oder gelöscht werden. Das Schnittstellenelement wird nur dann angezeigt, wenn *WS_VISIBLE* gesetzt ist.

Ein Element kann jederzeit angezeigt oder verborgen werden, indem man die *Show*-Elementfunktion des Schnittstellenobjekts aufruft.

Das Verhältnis zwischen über- und untergeordneten Fenstern

In einer Windows-Anwendung sind Schnittstellenelemente (Fenster, Dialogfenster und Steuerelemente) über eine hierarchische Beziehung miteinander verbunden. Zwei Schnittstellenelemente sind miteinander verknüpft, wenn das eine das übergeordnete und das andere dessen untergeordnetes Fenster ist. Diese Beziehung darf nicht mit Vererbung oder mit dem Zugehörigkeitsverhältnis von Instanzen verwechselt werden, bei denen es sich um Objektbeziehungen »bürokratischer« Art handelt. Ein untergeordnetes Fenster erbt nichts von seinem übergeordneten Fenster.

Ein untergeordnetes Fenster ist ein Schnittstellenelement, das von einem anderen Schnittstellenelement verwaltet wird. Beispielsweise werden Listen von dem Fenster bzw. von dem Dialogfenster verwaltet, in dem sie erscheinen. Sie werden nur dann angezeigt, wenn auch ihr übergeordnetes Fenster angezeigt wird. Dialogfenster sind untergeordnete Fenster und werden von dem Fenster verwaltet, das sie hervorbringt. Wird das übergeordnete Fenster geschlossen oder verschoben, so werden die untergeordneten Fenster automatisch geschlossen bzw. verschoben.

Alle Schnittstellenelemente (Fenster, Dialogfenster und Steuerelemente) können sowohl übergeordnete als auch untergeordnete Fenster sein.

Listen untergeordneter Fenster

Das übergeordnete Fenster eines Schnittstellenelements wird bei seiner Erstellung angegeben. Das übergeordnete Fensterobjekt ist ein Parameter des Schnittstellenobjekt-Konstruktors. Für den Zugriff auf das übergeordnete Fenster speichert das Objekt eines untergeordneten Fensters einen Zeiger auf das Objekt in seinem Datenelement *Parent*. Auf seine eigenen untergeordneten Fenster greift es über eine intern verwaltete Liste zu.

Wenn Sie eine neue Schnittstellenelement-Klasse mit untergeordneten Fenstern definieren, müssen Sie den Konstruktor so definieren, daß er auch die untergeordneten Fensterobjekte erstellt. Das Schnittstellenelement des übergeordneten Fensters wird erzeugt, wenn die Anwendung dessen *Create*-Elementfunktion aufruft. Bei erfolgreicher Ausführung wird die *SetupWindow*-Elementfunktion aufgerufen. *TWindowsObject::SetupWindow* iteriert über jedes Fenster in seiner intern geführten Liste der untergeordneten Fenster und ruft per Voreinstellung für jedes – mit Ausnahme von Dialogfenstern – dessen *Create*-Elementfunktion auf. Untergeordnete Fenster mit dem Status *WS_VISIBLE* werden sofort nach ihrer Erstellung angezeigt.

Es ist oft nützlich, die *SetupWindow*-Elementfunktion des übergeordneten Fensters zu überschreiben, um nach der Erzeugung der untergeordneten Fenster einige Einstellungen vornehmen zu können. Das Füllen einer Liste ist ein gutes Beispiel hierfür. In diesem Fall muß die erste Zeile der überschriebenen *SetupWindow*-Elementfunktion einen Aufruf der *SetupWindow*-Elementfunktion der Basisklasse Ihres Fensterobjekts enthalten. Ein Beispiel für das Überschreiben von *SetupWindow* finden Sie in Kapitel 6 »Fenster anzeigen«.

Um ein untergeordnetes Fenster (zum Beispiel ein Pop-up-Fenster) explizit von dieser Standardbehandlung auszuschließen, müssen Sie nach der Erstellung des Objekts die Elementfunktion *DisableAutoCreate* aufrufen. Wenn Sie für ein untergeordnetes Fenster (etwa ein Dialogfenster, das normalerweise ausgeschlossen würde) die Bearbeitung explizit anfordern, muß nach der Erstellung des Objekts die Elementfunktion *EnableAutoCreate* aufgerufen werden.

Wie der Aufruf der *Create*-Elementfunktion des übergeordneten Fensters zu einem Aufruf der *Create*-Elementfunktionen seiner untergeordneten Fenster führt, erzeugt auch der Aufruf des Destruktors des übergeordneten Fensters Aufrufe der Destruktoren seiner untergeordneten Fenster. Die *CanClose*-Elementfunktion iteriert ebenfalls über die Liste der untergeordneten Fenster und ruft deren *CanClose*-Elementfunktion auf. Sie liefert nur dann *TRUE* zurück, wenn alle untergeordneten Fenster *TRUE* zurückliefern.

Über untergeordnete Fenster iterieren

Manchmal werden Elementfunktionen benötigt, die über jedes untergeordnete Fenster iterieren und dabei eine Funktion ausführen.

Um zum Beispiel alle untergeordneten Auswahlfelder eines Fensters zu überprüfen, wird die Elementfunktion *ForEach*, die von *TWindowsObject* geerbt ist, folgendermaßen eingesetzt:

```
void CheckTheBox(Pvoid P, Pvoid Param)
{
  (PTCheckBox)P->Check();
}

void TMyWindow::CheckAllBoxes();
{
  ForEach(CheckTheBox, NULL);
}
```

Sie können auch Elementfunktionen schreiben, die über die Liste der untergeordneten Fenster iterieren und ein bestimmtes Fenster heraussuchen. Um beispielsweise aus vielen untergeordneten Auswahlfeldern das erste markierte zu finden, kann die Elementfunktion *FirstThat*, die von *TWindowsObject* geerbt ist, folgendermaßen eingesetzt werden:

```
BOOL IsThisBoxChecked(Pvoid P, Pvoid Param)
{
  return (PTCheckBox)P->GetCheck() == BF_CHECKED;
}

PTCheckBox TMyWindow::GetFirstChecked();
{
  return FirstThat(IsThisBoxChecked, NULL);
}
```

Botschafts-bearbeitung

ObjectWindows-Programme kommunizieren mit Windows über zwei Kanäle. In der einen Richtung steuert die Anwendung die Aktionen der Anwenderschnittstelle durch den Aufruf von Windows-Funktionen, in der anderen Richtung sendet Windows Botschaften als Reaktion auf *Ereignisse* im Programm, etwa der Auswahl eines Menüpunktes (*WM_COMMAND*) oder dem Drücken der linken Maustaste (*WM_LBUTTONDOWN*), an die Anwendung. Es existieren über 100 Standard-Windows-Botschaften. Zusätzlich können Sie Ihre eigenen Botschaften definieren (vgl. Kapitel 7 »Überblick über ObjectWindows«).

Windows-Anwendungen erledigen alle ihre Aufgaben durch die Beantwortung der Botschaften, die sie von Windows erhalten. Das anwendungsspezifische Verhalten wird durch selektives Beantworten der Botschaften erreicht. Dabei können Windows-Funktionen aufgerufen werden, die wiederum neue Windows-Botschaften generieren können. Eingehende Botschaften sind immer für ein spezifisches Fenster bestimmt – das Fenster, das vom aktuellen Ereignis direkt betroffen ist. Wenn beispielsweise in einem bestimmten Fenster die linke Maustaste gedrückt wird, »gehört« die erzeugte *WM_LBUTTONDOWN*-Botschaft zu diesem Fenster. ObjectWindows leitet die empfangenen Botschaften für Ihre Fenster an die entsprechenden Botschaftsantwortfunktionen weiter.

Das Zusammenspiel von Windows-Botschaften und Objekt-Elementfunktionen bildet das Grundgerüst einer ObjectWindows-Anwendung. Die Hauptaufgabe einer ObjectWindows-Anwendung besteht demnach in der Beantwortung von Windows-Ereignissen. Und hierin besteht auch der große Unterschied zur traditionellen programmgesteuerten DOS-Anwendung. Windows-Anwendungen erfordern eine auf Botschaften basierende, ereignisgesteuerte Programmierung (unter dem Motto »Erst sprechen, wenn Sie angesprochen werden«). Diese Art der Programmierung erfordert erhöhte Disziplin, liefert dafür aber intuitiv bedienbare und multitaskingfähige Anwendungen.

Botschaften beantworten

Damit Fensterobjekte auf eingehende Windows-Botschaften antworten, muß für jede Botschaft, die beantwortet werden soll, eine eigene Elementfunktion definiert werden. Derartige Elementfunktionen werden als *Botschaftsantwortfunktionen* bezeichnet. Ihre Erstellung wird durch eine Erweiterung der virtuellen Funktionsdeklaration erleichtert. Um eine Elementfunktion als Botschaftsantwortfunktion zu kennzeichnen, wird die Summe aus *WM_FIRST* und dem Botschaftsnamen an das Ende der virtuellen Elementfunktions-Deklaration angehängt:

```
class TMyWindow : public TWindow
{
public:
.
```

```
.
.
  virtual void WMRButtonDown(RTMessage Msg) =
    [WM_FIRST + WM_RBUTTONDOWN];
.
.
.
};
```

Hinweis: Nähere Informationen über Botschaftsanwortfunktionen finden Sie in Kapitel 1 »Das Fenster vererben«.

In diesem Beispiel wird die *WMRButtonDown*-Elementfunktion aufgerufen, wenn ein *TMyWindow*-Objekt eine *WM_RBUTTONDOWN*-Botschaft empfängt. Elementfunktions- und Botschaftsnamen müssen sich zwar nicht ähneln, die Lesbarkeit des Programms läßt sich aber dadurch verbessern. ObjectWindows verwendet diese Konvention in den Schnittstellenklassen.

Msg ist ein erforderliches Argument vom Typ *RTMessage*. *RTMessage* ist ein von ObjectWindows definierter Typ, der eine Referenz auf eine *TMessage*-Struktur mit den Feldern *WParam*, *LParam* und *Result* darstellt. Die Elemente *WParam* und *LParam* (die vom Typ *WORD* und *LONG* sind) beinhalten Informationen über die dem eingetretenen Ereignis entsprechende Botschaft. Häufig speichert jeder Parameter zwei Arten von Informationen. Wenn beispielsweise *WM_BUTTONDOWN* gesendet wird, enthält das niederwertige *WORD* von *LParam* die x-Koordinate und das höherwertige *WORD* die y-Koordinate des Punktes, an dem geklickt wurde.

LParam ist als Element einer Variante mit einer *LP*-Struktur deklariert, die *Hi* und *Lo* als Elemente vom Typ *WORD* enthält. Sie können aus diesem Grund mit *Msg.LP.Hi* und *Msg.LP.Lo* einen Bezug auf die höher- und niederwertigen *WORD*-Werte von *Msg.LParam* herstellen. *WParam* ist ähnlich als Element einer Variante mit einer *WP*-Struktur deklariert. Am häufigsten werden Sie die Felder *WParam*, *LP.Lo* und *LP.Hi* verwenden, die alle vom Typ *WORD* sind. *Msg* wird als Referenz übergeben, weil einige wenige Windows-Botschaften eine Antwort benötigen, die Ihre Elementfunktionen im *Result*-Element speichern.

Windows-Botschaften, auf die Ihre Objekte nicht antworten sollen, können Sie ignorieren. Sie brauchen in einem derartigen Fall keine Elementfunktion zu definieren, die auf diese Botschaften antwortet. Um diese Botschaften kümmert sich dann die Elementfunktion *DefWndProc*, die jede Fensterklasse von *TWindowsObject* erbt.

Für die Beantwortung einiger Botschaften, die im Zusammenhang mit Menüs und untergeordneten Fenstern häufig vorkommen (z.B. Editierfelder und Bildlaufleisten), besteht die Möglichkeit, Kürzel zu verwenden. Näheres erfahren Sie in den folgenden Abschnitten.

Befehlsabhängige Botschaften und Botschaften untergeordneter Fenster

Windows erzeugt eine *befehlsabhängige Botschaft*, wenn der Anwender einen Menüpunkt auswählt oder ein Tastenkürzel eingibt. Aktiviert er dagegen ein Steuerelement, zum Beispiel einen Schalter oder eine Liste, generiert Windows

eine Botschaft, die von der ID des untergeordneten Fensters abhängig ist. Die meisten derartigen Aktionen erzeugen eine *WM_COMMAND*-Botschaft.

ObjectWindows antwortet auf *WM_COMMAND*-Botschaften durch die automatische Erzeugung eines spezialisierteren, virtuellen Elementfunktionsaufrufs, der auf dem Inhalt der jeweiligen Botschaft basiert. Sie können sich daher das Schreiben umfangreicher **case**-Anweisungen sparen. Bei der Definition von Botschaftsantwortfunktionen zur Beantwortung von Botschaften, die von Befehlen oder der ID von untergeordneten Fenstern abhängig sind, kann eine Erweiterung der Elementfunktionsdefinition – ähnlich der bereits beschriebenen – benutzt werden.

Befehlsabhängige Botschaften bearbeiten

Verwenden Sie für die Verknüpfung eines Menü- oder Tastenkürzelereignisses mit einer Botschaftsantwortfunktion zur Beantwortung befehlsabhängiger Botschaften einen Bezeichner, der aus der Summe zweier Konstanten besteht. Die erste Konstante ist die in ObjectWindows definierte Konstante *CM_FIRST*, die zweite ist die Menü- oder Tastenkürzel-ID, die in der Menü- bzw. Tastenkürzelressource definiert ist und in der *WM_COMMAND*-Botschaft übergeben wird. Angenommen, ein Fenster (es sei ein *TFileWindow*-Objekt) enthält die drei Menüpunkte *File/New*, *File/Open* und *File/Save*. Die in der Ressourcendatei owlrc.h definierten Menü-IDs lauten *CM_FILENEW*, *CM_FILEOPEN* und *CM_FILESAVE*. Die Objektdefinition von *TFileWindow* könnte dann folgendermaßen aussehen:

```
class TFileWindow : public TWindow
{
.
.
.
  virtual void CMFileNew(RTMessage Msg) =
    [CM_FIRST + CM_FILENEW];
  virtual void CMFileOpen(RTMessage Msg) =
    [CM_FIRST + CM_FILEOPEN];
  virtual void CMFileSave(RTMessage Msg) =
    [CM_FIRST + CM_FILESAVE];
.
.
.
}
```

Ein Fenster mit einem Menü muß aber nicht immer eine Antwort auf befehlsabhängige Botschaften definieren. Diese Aufgabe kann auch ein untergeordnetes Fenster (auch ein Steuerelement) übernehmen. Das untergeordnete Fenster, das zum Zeitpunkt der Menüauswahl bzw. der Tastenkürzeleingabe den Fokus besitzt, hat die erste Gelegenheit zur Beantwortung einer befehlsabhängigen Botschaft durch die Definition einer Botschaftsantwortfunktion. Wenn es keine Botschaftsantwortfunktion definiert hat, wird die Botschaft an sein übergeordnetes Fenster, an das diesem Fenster übergeordnete Fenster, usw. weitergegeben, bis eine Antwort oder das Fenster gefunden wird, zu dem das

Menü gehört. Das bedeutet, daß ein Steuerelement in einem Fenster direkt auf eine Menüauswahl antworten kann. Ein Beispiel für diese Technik ist die ObjectWindows-Klasse *TEditWindow*.

Botschaften untergeordneter Fenster bearbeiten

Beeinflußt der Anwender etwa durch das Anklicken eines Schalters oder die Eingabe in ein Editierfeld ein Steuerelement oder ein sonstiges untergeordnetes Fenster, erhält das übergeordnete Fenster oder das Dialogobjekt eine *Benachrichtigungsbotschaft* vom untergeordneten Fenster. Damit das übergeordnete Fenster auf eine Benachrichtigungsbotschaft antwortet, definieren Sie eine Elementfunktion mit einem Bezeichner, der sich aus der Summe zweier Konstanten zusammensetzt: der in ObjectWindows definierten Konstante *ID_FIRST* und der ID des untergeordneten Fensters, die in einer Dialogressource oder bei der Erstellung eines Steuerelements definiert wird.

Enthält beispielsweise ein Fenster eine Liste, kann zur Beantwortung der von der Liste generierten, auf der ID des untergeordneten Fensters basierenden Botschaften eine Elementfunktion mit dem Namen *HandleListBoxMsg* definiert werden:

```
class TListBoxWindow : public TWindow
{
.
.
.
  virtual void HandleListBoxMsg(RTMessage Msg) =
    [ID_FIRST + ID_LISTBOX];
}
```

Hierbei ist *ID_LISTBOX* eine Konstante, die die Steuerelement-ID repräsentiert. Als ID von untergeordneten Fenstern sind nur positive ganze Zahlen zulässig, die kleiner als 4096 sind.

Alternativ zur Botschaftsbeantwortung durch das übergeordnete Fenster kann die Antwort auch direkt über das Steuerelement durch Definition einer auf Benachrichtigung basierenden Botschaftsantwortfunktion erfolgen. Näheres hierzu finden Sie im Abschnitt »Benachrichtigungsbotschaften der Steuerelemente beantworten« in Kapitel 12 »Steuerelementobjekte«.

Botschaftsbearbeitung durch Standardantworten

Dieser Abschnitt befaßt sich mit Windows-Botschaften, die direkt abgefangen und mit Hilfe der Konstanten *WM_FIRST* (Windows-Botschaften) berechnet werden, die aber nicht von Befehlen oder von der ID untergeordneter Fenster abhängig und keine Benachrichtigungsbotschaften sind. Informationen zu auf Benachrichtung basierenden Botschaften finden Sie in Kapitel 12 »Steuerelementobjekte«.

Eine Windows-Anwendung empfängt mehr Botschaften, als beantwortet werden sollen bzw. müssen. Windows stellt Standardantworten für viele dieser Botschaften zur Verfügung, vor allem für Botschaften an Steuerelemente. Gibt

der Anwender beispielsweise Text in ein Editierfeld ein, antwortet Windows automatisch mit der Anzeige der Zeichen und (falls erforderlich) mit dem Bildlauf des Textes. Der Aufruf dieser Standardantworten für ObjectWindows-Fenster erfolgt durch die Elementfunktion *DefWndProc*.

Wenn Sie eine Windows-Botschaft ignorieren (d.h. keine Botschaftsantwortfunktion definieren), ruft ObjectWindows automatisch die Elementfunktion *DefWndProc* des Schnittstellenobjekts auf. Der Aufruf von *DefWndProc* kann aber auch dann erforderlich sein, wenn Sie die Botschaft durch die Definition einer Botschaftsantwortfunktion abfangen.

Zum Beispiel fängt das folgende Programmfragment die Botschaft *WM_CHAR* ab, die nach einer Eingabe des Anwenders in ein Editierfeld an das entsprechende Editierfeld-Objekt gesendet wird. Die von *TEdit* abgeleitete Botschaftsantwortfunktion für *WM_CHAR* sieht folgendermaßen aus:

```
void TMyEdit::WMChar(RTMessage Msg)
{
  MessageBeep(0);
}
```

Diese Elementfunktion antwortet auf die Eingabe des Anwenders mit einem Signalton. Die Definition einer Antwort verhindert aber die Standardantwort der voreingestellten Fensterprozedur. Das hat zur Folge, daß zwar der Signalton ertönt, der eingegebene Text aber nicht im Editierfeld angezeigt wird.

Damit der Signalton ertönt *und* die Standardantwort ausgelöst wird, muß die *DefWndProc*-Elementfunktion explizit aufgerufen werden:

```
void TMyEdit::WMChar(RTMessage Msg)
{
  MessageBeep(0);
  DefWndProc(Msg);
}
```

Diese Elementfunktion sorgt dafür, daß der Signalton ertönt und das eingegebene Zeichen in das Editierfeld eingefügt wird.

DefWndProc muß in folgenden Fällen aufgerufen werden:

- Wenn Sie wissen wollen, ob eine Botschaft empfangen wurde, aber die Standardbearbeitung nicht überschrieben werden soll.
- Wenn eine Botschaft eine andere Aktion als die Standardantwort auslösen soll, die Standardantwort aber trotzdem benötigt wird.
- Wenn das Abfangen einer Botschaft die erwartete Aktion außer Kraft zu setzen scheint.

10. Fensterobjekte

In diesem Kapitel erfahren Sie, wie die Fenster einer Anwendung erstellt, angezeigt und ausgefüllt werden. Die ObjectWindows-Klasse *TWindow* definiert das grundlegende Verhalten und die Eigenschaften des Hauptfensters und der untergeordneten Fenster. *TWindow* definiert das Verhalten und die Eigenschaften für das Öffnen, Schließen, Zeichnen und den Bildlauf von Fenstern und untergeordneten Fenstern sowie für die Bearbeitung von Befehlsbotschaften.

Von *TWindow* stammen vier Klassen ab: *TMDIFrame*, *TMDIClient*, *TControl* und *TEditWindow*. *TMDIFrame* und *TMDIClient* werden in ObjectWindows in den Anwendungen eingesetzt, die dem Standard der Multidokumentschnittstelle (MDI) entsprechen (MDI steht als Abkürzung für »Multiple Documents Interface«). Eine Erläuterung der MDI und der dazugehörigen Typen finden Sie in Kapitel 14 »MDI-Objekte«. *TControl* definiert Steuerelemente wie Schalter und Listen und wird in Kapitel 12 »Steuerelementobjekte« erläutert. Am häufigsten werden Sie jedoch Fensterklassen anlegen, die von *TWindow* abgeleitet sind.

In diesem Kapitel werden die Typen *TWindow*, *TEditWindow* und *TFileWindow* erläutert. Außerdem finden Sie ein Beispiel für die Registrierung neuer Windows-Klassen.

Die Klasse TWindow

Eine ObjectWindows-Anwendung muß zumindest über ein Hauptfenster verfügen, das beim Start der Anwendung auf dem Bildschirm angezeigt wird. Normalerweise ist das Hauptfenster eines ObjectWindows-Programms eine Instanz einer von *TWindow* abgeleiteten Klasse. Viele Anwendungen haben weitere Fenster, die meistens untergeordnete Fenster des Hauptfensters sind.

Diese zusätzlichen Fenster sind ebenfalls Instanzen einer von *TWindow* abgeleiteten Klasse. So kann ein Zeichenprogramm beispielsweise für sein Hauptfenster den Typ *TPaintWindow* und für ein anderes Fenster den Typ *TZoomWindow* definieren, der eine vergrößerte Darstellung des Gezeichneten anzeigt. In diesem Fall stammen beide Typen von *TWindow* ab. In einer gut gestalteten Anwendung wäre *ZoomWindow* eine spezielle Version von *TPaintWindow* und als eine von *TPaintWindow* (also indirekt von *TWindow*) abgeleitete Klasse implementiert.

Fensterobjekte initialisieren und erstellen

Wie Dialog- und Steuerelementobjekte sind Fensterobjekte Schnittstellenobjekte, die mit visuellen Schnittstellenelementen verbunden sind. Genauer gesagt repräsentieren sie Fensterelemente, die durch ein Handle verknüpft sind, das in ihrem Datenelement *HWindow* gespeichert ist. Da ein Fenster aus zwei Teilen besteht, wird dieses Fenster in zwei Schritten erstellt und zwar in folgender Reihenfolge: Initialisieren des Objekts und Erstellen des visuellen Elements.

Ein Fensterobjekt initialisieren

Eine typische Windows-Anwendung verfügt über viele unterschiedliche Fenstertypen: überlappende oder Pop-up-, gerahmte, rollbare oder betitelte Fenster und viele mehr. Die Attribute für diese Stile werden – ebenso wie andere Erstelltungsattribute – während der Initialisierung eines Fensterobjekts bestimmt und während des Erstellens seines visuellen Elements verwendet.

Die Erstellungsattribute eines Fensterobjekts, wie Stil, Titel und Menü, werden in seinem Datenelement *Attr* gespeichert, das als Typ *TWindowAttr* definiert ist. *TWindowAttr* setzt sich aus folgenden Datenelementen zusammen:

Tabelle 10.1 TWindowAttr-Datenelemente

Datenelement	Verwendung
Style	Ein DWORD, das die bitweise verknüpften Stilkonstanten enthält.
ExStyle	Ein DWORD, das den erweiterten Stil enthält.
Menu	Ein *LPSTR*, der die Menüressource bezeichnet.
X	Ein Integer, der die horizontale Startposition des Fensters enthält. Dabei handelt es sich um die horizontale Bildschirmkoordinate der linken oberen Fensterecke.
Y	Ein Integer, der die vertikale Startposition des Fensters enthält. Dabei handelt es sich um die vertikale Bildschirmkoordinate der linken oberen Fensterecke.
W	Ein Integer, der die anfängliche Fensterbreite in Bildschirmkoordinaten angibt.
H	Ein Integer, der die anfängliche Fensterhöhe in Bildschirmkoordinaten angibt.
Param	Ein *LPSTR*, der bei der Erstellung an das Fenster übergeben wird.
Id	Ein Integer, der die ID des untergeordneten Fensters angibt. Diese wird für die Kommunikation zwischen einem Steuerelement und dessen übergeordneten Fenster oder Dialog benötigt. *Id* muß für alle untergeordneten Fenster desselben übergeordneten Fensters eindeutig sein. Ist das Steuerelement in einer Ressource definiert, muß seine *Id* mit der ID der Ressource übereinstimmen. Für ein Fenster darf nicht gleichzeitig *Menu* und *Id* gesetzt sein.

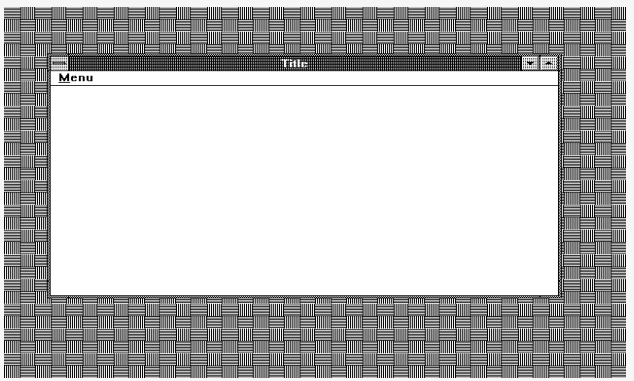


Abbildung 10.1
Fensterattribute

Das erste Argument im Konstruktor-Aufruf ist das Fensterobjekt des übergeordneten Fensters. Es ist *NULL*, wenn es sich bei dem Fenster um das Hauptfenster (das kein übergeordnetes Fenster hat) handelt. Der *TWindow*-Konstruktor setzt das *Title*-Datenelement von *Attr* auf das angegebene *LPSTR*-Argument. Per Voreinstellung wird das *Style*-Datenelement von *Attr* auf *WS_OVERLAPPEDWINDOW* gesetzt, wenn es sich bei dem Fenster um das Hauptfenster der Anwendung handelt (andernfalls *WS_VISIBLE*). *X* und *W* werden auf *CW_USEDEFAULT* und *Y* und *H* auf 0 gesetzt. Das Ergebnis ist ein überlappendes Fenster normaler Größe. Das ist der normale Weg, um die Größe eines neuen Hauptfensters festzulegen.

Wenn Sie von *TWindow* neue Fensterklassen ableiten, müssen Sie einen Konstruktor definieren, der (als Mindestaufgabe) den Konstruktor seiner Basisklasse aufruft:

```
class MyWindow : public TWindow
{
  ...
  TMyWindow(PTWindowsObject AParent, LPSTR ATitle) :
            TWindow(AParent, ATitle) {}
  ...
};
```

Sie können dann die Attribute für Ihr Fensterobjekt neu bestimmen, indem Sie das *Attr*-Datenelement direkt im Konstruktor ändern. Sie können das Fenster beispielsweise vergrößert anzeigen:

```
class TMyWindow : public TWindow
{
  ...
  TMyWindow(PTWindowsObject AParent, LPSTR ATitle);
  ...
  PTChildWindow AChildWindow;
  PTListBox ListBox;
};
...
```

```
TMyWindow::TMyWindow(PTWindowsObject AParent, LPSTR ATitle) :
                     TWindow(AParent, ATitle)
{
  Attr.Menu |= WS_MAXIMIZE;
  AChildWindow = new TWindow(this, "Child Title");
  ListBox = new TListBox(this, ID_LISTBOX, 201, 20, 20, 180, 80);
  ...
}
```

Beachten Sie, daß im obigen Beispiel der Fenster-Konstruktor für die Initialisierung der untergeordneten Fenster zuständig ist, wie zum Beispiel für Pop-up-Fenster und Listen. Umgekehrt kann eine untergeordnete Fensterklasse ihre Attribute in ihrem eigenen Konstruktor neu setzen:

```
TChildWindow::TChildWindow(PTWindowsObject AParent,
                           LPSTR ATitle) : TWindow(AParent, ATitle)
{
  Attr.Style |= WS_POPUP | WS_CAPTION;
  Attr.X = 100;
  Attr.Y = 100;
  Attr.W = 300;
  Attr.H = 300;
}
```

Das Attribut *Style* kann eine Fensterstilkonstante oder eine Kombination mehrerer Stilkonstanten sein, wie beispielsweise

```
WS_OVERLAPPEDWINDOW
WS_POPUPWINDOW
WS_CHILDWINDOW
WS_CAPTION
WS_BORDER
WS_VSCROLL
```

Diese Liste enthält einige gebräuchliche Stilkonstanten. Eine Zusammenstellung aller Konstanten finden Sie in Kapitel 4, »Windows-Stile und -Konstanten« im *Windows-Referenzhandbuch*. Wollen Sie keine abgeleitete Fensterklasse für ein untergeordnetes Fenster definieren, können Sie das Fensterobjekt erzeugen und dann dessen Attribute im Konstruktor des übergeordneten Fensters setzen:

```
TMyWindow::TMyWindow(PTWindowsObject AParent, LPSTR ATitle) :
                     TWindow(AParent, ATitle)
{
  AssignMenu(100)
  AChildWindow = new TChildWindow(this, "Child Title");
  AChildWindow->Attr.Style = WS_POPUP | WS_CAPTION;
  AChildWindow->Attr.X = 100;
  AChildWindow->Attr.Y = 100;
  AChildWindow->Attr.W = 300;
  AChildWindow->Attr.H = 300;
...
}
```

Hinweis: Die Elementfunktion *AssignMenu* wird verwendet, da das *Attr.Menu*-Element nicht direkt gesetzt werden darf.

Fensterelemente erstellen

Das Erstellen von Fensterelementen ist der Aufbauprozeß von visuellen Elementen, die mit einem Fensterobjekt verknüpft sind. Dieser Aufbau erfolgt durch Aufruf der Elementfunktion *MakeWindow* des Anwendungsobjekts, der das Fensterobjekt als Parameter übergeben wird:

```
if (GetApplication() -> MakeWindow(AWindow))
  // Erstellung erfolgreich
else
  // Erstellung nicht erfolgreich
```

GetApplication() liefert einen Zeiger auf das Anwendungsobjekt zurück. *MakeWindow* ruft zwei wichtige Elementfuntkionen auf. Die erste trägt den Namen *ValidWindow* und überprüft, ob das Fensterobjekt erfolgreich initialisiert wurde. Wenn der Konstruktor aus irgendeinem Grund nicht erfolgreich war, liefert *MakeWindow NULL* zurück und entfernt das Fensterobjekt. War der Konstruktor erfolgreich, setzt *MakeWindow* seine Arbeit fort und ruft die *Create*-Elementfunktion auf, die Windows dazu veranlaß t, ein visuelles Element zu erstellen. Wenn *Create* nicht erfolgreich ist, löscht *MakeWindow* das Fensterobjekt und liefert *NULL* zurück. Andernfalls wird ein Zeiger auf das Fensterobjekt zurückgegeben.

Obwohl *Create* die Elementfunktion ist, die das Fensterelement tatsächlich erstellt, werden Sie *Create* normalerweise nicht explizit aufrufen. Das Hauptfenster der Anwendung wird beim Start der Anwendung durch *TApplication::InitInstance* erstellt. Alle anderen Fenster in der Anwendung sind untergeordnete Fenster, die direkt oder indirekt vom Hauptfenster abstammen. Untergeordnete Fenster werden normalerweise in der *SetupWindow*-Elementfunktion ihrer übergeordneten Fensterobjekte oder dynamisch mit *MakeWindow* erstellt.

Zusammenfassung

Wenn Sie eine neue Fensterklasse entwerfen, können Sie deren Konstruktor überschreiben, so daß die Fensterattribute gesetzt und die untergeordneten Fensterobjekte (falls vorhanden) erzeugt werden. Die Attribute eines Fensters können ebenfalls im Konstruktor des übergeordneten Fensters gesetzt werden. *GetApplication()->MakeWindow* baut ein visuelles Element für das Fensterobjekt auf. Zu diesem Zweck wird *Create* – die Elementfunktion des Fensters – aufgerufen. Wenn sich unter den Stilen im Datenelement *Attr.Style WS_VISIBLE* befindet, wird das Fenster angezeigt, sobald es erstellt ist. Danach wird die Elementfunktion *SetupWindow* aufgerufen, die per Voreinstellung die untergeordneten Fenster erstellt.

Der Konstruktor des übergeordneten Fensters erzeugt normalerweise die untergeordneten Fenster. Die Attribute eines Fensterobjekts werden normalerweise von seinem Konstruktor oder vom Konstruktor seines übergeordneten Fensters gesetzt. Da das Hauptfenster einer Anwendung kein übergeordnetes Fenster hat, wird es vom Anwendungsobjekt beim Start der Anwendung initialisiert und erstellt.

Windows-Fensterklassen-Registrierung

Wie Sie gesehen haben, können bei der Initialisierung von Fensterobjekten die Attribute des Fensters (wie Stil, Position und Menü) durch Ausfüllen des Datenelements *Attr* bestimmt werden (das Menü des Fensters können Sie durch einen Aufruf der *AssignMenu*-Elementfunktion aktualisieren). Diese Attribute werden beim Erstellen des entsprechenden Fensterelements verwendet und daher *Erstellungsattribute* genannt.

Es gibt jedoch noch einige weitere Attribute, wie Hintergrundfarbe, Symbol und Mauszeiger. Sie können sich die verschiedenen Attribute ansehen, indem Sie gleichzeitig unterschiedliche Anwendungen in Windows ablaufen lassen. Wenn Sie den Mauszeiger von einer Anwendung in eine andere bewegen, wechselt der Zeiger sein Aussehen: aus dem (regulären) Pfeil wird ein dickes Kreuz bei Kalkulationsprogrammen und ein dünnes Kreuz bei Grafikprogrammen.

Diese »anderen« Attribute eines Fensters (einschließlich seines Symbols) werden durch Bezeichnung der Windows-Fensterklasse des Fensters bei seiner Erstellung bestimmt. Die für eine Windows-Fensterklasse definierten Attribute können nicht – wie die Erstellungsattribute – für jedes Fenster geändert werden, sondern gelten für alle Fenster mit derselben Windows-Fensterklasse

Die Windows-Fensterklasse eines Fensters ist nicht identisch mit der ObjectWindows-Fensterklasse in C++. Jede Instanz einer ObjectWindows-Fensterklasse gehört jedoch zu derselben Windows-Fensterklasse.

Eine Windows-Fensterklasse muß mit Windows *registriert* werden, bevor ein Fenster mit dieser Fensterklasse erzeugt wird. ObjectWindows führt diesen Registrierungsprozeß bei der Erstellung der Fenster selbständig durch. ObjectWindows stellt eine Standard-Windows-Fensterklasse mit voreingestellten *Registrierungsattributen* zur Verfügung. Wenn Sie keines dieser Attribute ändern wollen, müssen Sie sich um die Registrierung überhaupt nicht kümmern.

Möchten Sie eines dieser Registrierungsattribute für eines Ihrer Fenster ändern, müssen Sie das Fenster mit einer neuen Windows-Fensterklasse, die Sie definieren, verknüpfen. Dazu überschreiben Sie zwei Elementfunktionen Ihres Fensters: *GetWindowClass* und *GetClassName*. In *GetWindowClass* geben Sie die gewünschten Registrierungsattribute an. Durch Überschreiben von *GetClassName* geben Sie der Windows-Fensterklasse einen Namen. *GetClassName* ist eine Funktion, die einfach den Namen der Fensterklasse (*LPSTR*) zurückliefert. *TWindowsObject* definiert eine *GetClassName*-Elementfunktion, die "OWLWindow", den Namen der voreingestellten Windows-Fensterklasse, zurückliefert:

Zur Definition einer Windows-Fensterklasse namens *IBeamWindow*, die anstelle des Standardpfeils einen I-Balkenzeiger verwendet, überschreiben Sie die geerbte *GetClassName*-Elementfunktion folgendermaßen:

```
LPSTR IBeamWindow::GetClassName()
{
  return "IBeamWindow";
}
```

Der zurückgelieferte Name der Klasse kann – muß aber nicht – derselbe sein wie der tatsächliche C++-Klassenname. Der zurückgelieferte Klassenname wird als Name für die Windows-Fensterklasse verwendet, nachdem sie für Windows registriert ist. Die Elementfunktion *GetWindowClass* hat eine *WNDCLASS*-Struktur als Referenzargument und füllt deren Felder mit den neuen Registrierungsattributen aus. Als erstes sollten Sie die Elementfunktion *GetWindowClass* der Basisklasse aufrufen und dann die Felder setzen, die geändert werden sollen. Im folgenden Beispiel wird *TWindow::GetWindowClass* aufgerufen, und dann das Feld *hCursor*, das ein Handle für eine Mauszeigerressource enthält, überschrieben:

```
void IBeamWindow::GetWindowClass(WNDCLASS& AWndClass)
{
  TWindow::GetWindowClass(AWndClass);
  AWndClass.hCursor = LoadCursor(NULL, IDC_IBEAM);
}
```

IDC_BEAM ist eine Konstante, die einen der vordefinierten Windows-Mauszeiger repräsentiert. In Abbildung 10.2 sehen Sie eine Anwendung, die das *IBeamWindow*-Objekt einsetzt.

Abbildung 10.2
Ein Programm, das die Klasse IBeamWindow benutzt

Registrierungsattribute

ObjectWindows definiert eine Windows-Fensterklasse, die per Voreinstellung für Instanzen von *TWindow* verwendet wird. "OWLWindow"-Registrie-

rungsattribute schließen ein leeres Symbol (ein weißes Rechteck), einen Pfeilzeiger und eine Standardfensterfarbe ein. Wenn Sie diese Charakteristika für Ihre von *TWindow* abgeleitete Klasse ändern wollen, müssen Sie ihre *GetWindowClass*-Elementfunktion überschreiben. *GetWindowClass* wird eine Referenz auf eine *WNDCLASS*-Struktur übergeben, die Sie mit Registrierungsattributen ausfüllen müssen.

In der folgenden Tabelle sehen Sie die Charakteristikafelder einer *WNDCLASS*-Struktur und die Standardwerte, auf die sie in *TWindow::GetWindowClass* gesetzt sind:

Tabelle 10.2 Registrierungsattribute

Charakteristika	Feld	Standardwert
Klassenstil	*Style*	*CS_HREDRAW | CS_VREDRAW*
Symbol	*hIcon*	*LoadIcon(0, IDI_APPLICATION)*
Mauszeiger	*hCursor*	*LoadCursor(0, IDC_ARROW)*
Hintergrundfarbe	*hbrBackground*	*(HBRUSH)(COLOR_WINDOW + 1)*
Standardmenü	*lpszMenuName*	*NULL*

Das Element für den Klassenstil

Das Element *Style* einer *WNDCLASS*-Struktur unterscheidet sich vom Fensterstil, der als *Erstellungsattribut* eines Fensterobjekts in *Attr.Style* festgelegt ist. *Attr.Style* wird beim Erstellen eines Fensters zur Festlegung der visuellen Erscheinungsform verwendet und mit *WS_*-KONSTANTEN (»Window-Style«) gesetzt.

Das *Style*-Element einer *WNDCLASS*-Struktur wird zur Festlegung bestimmter Verhaltensweisen und Eigenschaften eines Fensters nach dessen Erstellung verwendet. Es wird mit Hilfe der *CS_*-Konstanten (»Class-Style«) gesetzt. Wenn zum Beispiel *CS_HREDRAW* festgelegt ist, wird das gesamte Fenster neu gezeichnet, wenn sich seine Breite ändert. Ist *CS_NOCLOSE* gesetzt, wird die *Close*-Option im Systemmenü gesperrt. Das *Style*-Element kann mit einer *CS_*-Konstanten oder einer Kombination dieser Konstanten gefüllt werden.

Eine Liste aller *CS_*-Konstanten finden Sie in Kapitel 4 »Windows-Stile und -Konstanten« im *Windows-Referenzhandbuch*.

TWindow::GetWindowsClass definiert dieses Element folgendermaßen:

```
AWndClass.style = CS_HREDRAW | CS_VREDRAW;
```

Das Element für das Symbol

Das Element *hIcon* enthält ein Handle für ein Symbol, das zur Repräsentation eines Fensters in seinem verkleinerten Zusand dient. Normalerweise werden Sie eine Symbolressource definieren, die das Hauptfenster Ihrer Anwendung repräsentiert. *TWindow::GetWindowClass* setzt dieses Element auf das Handle des vordefinierten Symbols *IDI_APPLICATION*, das als weißes (leeres) Rechteck angezeigt wird.

Das Element für den Zeiger

Das Element *hCursor* enthält ein Handle für einen Mauszeiger, der verwendet wird, wenn der Mauszeiger im Fensterbereich steht. *TWindow::GetWindowClass*

setzt dieses Element auf das Handle des Standard-Windows-Pfeils *IDC_ARROW*. Auß erdem stehen die folgenden, vordefinierten Mauszeiger zur Verfügung:

```
IDC_IBEAM     IDC_SIZE        IDC_SIZENESW
IDC_WAIT      IDC_UPARROW     IDC_SIZEWE
IDC_CROSS     IDC_SIZENWSE    IDC_SIZENS
```

Mauszeiger können ebenso wie Symbole frei definiert werden.

Das Element für die Hintergrundfarbe

Das Element *hbrBackground* legt die Hintergrundfarbe für das Fenster fest. *TWindow::GetWindowClass* setzt dieses Element auf die Standardfensterfarbe, die der Anwender in der Systemsteuerung bestimmen kann. Sie können dieses Element mit dem Handle eines virtuellen Pinsels besetzen.

Alternativ dazu können Sie auch einen der Windows-Farbwerte angeben, wie zum Beispiel *COLOR_ACTIVECAPTION*, wobei *hbrBackground* auf *COLOR_ACTIVECAPTION* +1 gesetzt wird.

Das Element für das Standardmenü

Das Element *lpszMenuName* enthält einen Zeiger auf den Namen einer Menüressource, die von allen Fenstern der Windows-Fensterklasse verwendet wird. Wenn Sie beispielsweise einen *EditWindow*-Typ definiert haben, der immer über ein Standard-Editiermenü verfügen soll, so könnten Sie das betref - fende Menü hier festlegen. Dadurch ist es nicht mehr erforderlich, das Menü als Erstellungsattribut in einem Konstruktor festzulegen.

Hat die Menüressource eine ID von 101, können Sie das Feld folgendermaß en belegen:

```
AWndClass.lpszMenuName = MAKEINTRESOURCE(101);
```

Bildlauf

Sieht man durch ein Fenster, so ist das Blickfeld normalerweise begrenzt. Glücklicherweise können ObjectWindows-Fenster so programmiert werden, daß sie ihre Inhalte für den Betrachter verschieben. Die Klasse *TWindow* führt diese Aufgabe mit Hilfe eines *TScroller*-Objekts aus.

Meistens werden die Anwender die Bildlaufleisten an den Rändern des Client-Bereichs bedienen, um die aktuelle Anzeige zu verschieben. Diese Bildlaufleisten sind keine Steuerelemente, sondern Bestandteil des Fensters selbst, angelegt für Fenster, welche die Stilkonstanten *WS_VSCROLL* und/oder *WS_HSCROLL* enthalten. *TScroller* reagiert auf alle Botschaften für die Bildlaufleisten, so daß der Bildlauf nicht neu programmiert werden muß.

Bildlauf kann für ObjectWindows-Fenster auch dann durchgeführt werden, wenn der Anwender die Maus aus dem Client-Bereich des Fensters hinausbewegt. Diesen Mechanismus bezeichnet man als *automatischen Bildlauf*. Er funktioniert auch bei Fenstern, die keine Bildlaufleisten haben.

Scroller-Attribute

Ein *TScroller* enthält Werte für Schrittweiten, die festlegen, wieviel von einem Fensterinhalt verschoben werden soll. Diese Werte sind in den Datenelementen von *TScroller* gespeichert, also in *XUnit*, *YUnit*, *XLine*, *YLine*, *XPage*, *YPage*, *XRange* und *YRange*. Felder, die mit *X* beginnen, repräsentieren horizontale, Felder mit *Y* am Anfang vertikale Werte.

Eine Schrittweite bestimmt die kleinste Anzahl von Geräteeinheiten (d.h. Pixel in einem Fenster, abhängig vom eingestellten Abbildungsmodus), in der vertikal oder horizontal geblättert werden kann. Die Schrittweite hängt nomalerweise von der Art der Information ab, die angezeigt werden soll. Wenn Sie beispielsweise Text anzeigen, der eine Zeichenbreite von 8 und eine Zeichenhöhe von 15 Pixeln hat, sollten die Werte für *XUnit* und *YUnit* 8 bzw. 15 betragen.

Die weiteren Scroller-Attribute, *Line* (Zeile), *Page* (Seite) und *Range* (Bereich) werden in Schrittweiten ausgedrückt. *Line*- und *Page*-Werte geben die Anzahl der Schritte an, um die als Antwort auf eine Anforderung des Anwenders geblättert werden soll. Wenn der Anwender einen der Pfeile am Ende der Bildlaufleiste anklickt, wird der Fensterinhalt entsprechend der im Datenelement *Line* gespeicherten Anzahl von Schritten zeilenweise verschoben. Ein Klicken innerhalb der Bildlaufleiste selbst (nicht aber ein Anklicken des Positionszeigers) hat einen seitenweisen Bildlauf zur Folge. Die *Range*-Attribute repräsentieren die Gesamtanzahl der Schritte, um die verschoben werden kann. Dieser Wert beruht normalerweise auf der Größe der Informationen im Fenster, wie zum Beispiel dem editierten Dokument.

Sehen wir uns nun als Beispiel ein Fenster zum Editieren von Text an. Wenn Sie eine Textdatei anzeigen wollen, die 400 Textzeilen mit einer Breite von 80 Zeichen pro Zeile und 50 Zeilen pro Seite hat, sollten Sie sinnvollerweise die in Tabelle 10.3 dargestellten Werte festlegen.

Tabelle 10.3 Typische Belegung der Datenelemente eines Editierfensters

Datenelement	Wert	Bedeutung
XUnit	8	Zeichenbreite
YUnit	15	Zeichenhöhe
XLine, YLine	1	1 Schritt pro Textzeile
XPage	40	40 Zeichen pro horizontaler Seite
YPage	50	50 Zeilen pro vertikaler Seite
XRange	80	Maximaler horizontaler Bereich
YRange	400	Maximaler vertikaler Bereich

Ein *TScroller*-Objekt mit diesen Werten kann Bildlauf um eine Zeile oder eine Seite durchführen. Die gesamte Datei wird mit Hilfe der Bildlaufleisten oder des automatischen Bildlaufs angezeigt.

Der voreingestellte Zeilenwert beträgt 1 und muß nur in Ausnahmefällen geändert werden. Es gibt auch eine Standardformel für das Setzen der Schrittweite beim seitenweisen »Blättern«, ausgehend von der aktuellen Größe des

Fensters. In diesem Fall bedeutet der seitenweise Bildlauf ein Verschieben der Anzeige um die Höhe oder Breite des Client-Bereichs, abhängig von der Bildlaufrichtung. Deshalb ist es nicht notwendig, daß Sie die Seitenwerte ändern, außer, wenn Sie diesen Mechanismus überschreiben wollen.

Dem Fenster einen Scroller zuweisen

Soll Ihr Fenster über einen Scroller verfügen, erzeugen Sie ein *TScroller*-Objekt in Ihrem Fenster-Konstruktor und speichern einen Zeiger auf das Objekt im Fenster-Datenelement *Scroller*. Dabei können die Größe der Schrittweiten und der Bereich festgelegt werden, falls diese Werte bekannt sind. Obwohl Bildlaufleisten für die Verwendung eines Scrollers nicht benötigt werden (automatischer Bildlauf), verfügen viele bildlauffähige Fenster trotzdem darüber. Wollen Sie ein Fenster mit Bildlaufleisten versehen, fügen Sie nur *WS_HSCROLL* oder *WS_VSCROLL* (oder beide) in das *Attr.Style*-Datenelement des Konstruktors ein. Im folgenden sehen Sie ein Beispiel für einen Konstruktor des Fensters zum Editieren von Text (dieses Programmfragment ist Teil der Datei SCROLAPP.CPP):

```
TScrollWindow::TScrollWindow(LPSTR ATitle) : TWindow(NULL,
                                                     ATitle)
{
  Attr.Style |= WS_VSCROLL | WS_HSCROLL;
  Scroller = new TScroller(this, 8, 15, 80, 60);
}
```

Der T*Scroller*-Konstruktor hat als Argumente das bildlauffähige Fenster und die Anfangswerte für die Datenelemente *XUnit*, *YUnit*, *XRange* und *YRange*. Die Zeilen- und Seitenattribute sind auf deren Standardwerte gesetzt.

Bei Anzeige des Fensters kann der Inhalt des Client-Bereichs vertikal oder horizontal mit Hilfe der Bildlaufleisten oder über den automatischen Bildlauf verschoben werden. Die *Paint*-Elementfunktion des Fensters zeichnet die grafischen Informationen, ohne dabei berücksichtigen zu müssen, ob verschoben werden kann. Es ist selbstverständlich nicht effizient, ein großes Bild zu zeichnen, wenn nur ein Teil davon sichtbar ist. Die *Paint*-Elementfunktion kann jedoch dahingehend optimiert werden, daß nur die sichtbaren Teile des Bildes gezeichnet werden. Mehr zu diesem Punkt erfahren Sie am Ende dieses Abschnitts.

Ein Bildlauf-Beispiel

Wir werden Ihnen im folgenden Beispiel vorführen, wie eine vollständige Anwendung aufgebaut wird, die einen bildlauffähigen Grafikentwurf darstellt. Das Beispiel zeichnet eine Reihe von Rechtecken mit wachsender Größe, so daß das Bild nicht vollständig in den Client-Bereich eines Bildschirms mit einer gewöhnlichen VGA-Karte paßt. Mit Hilfe der Bildlaufleisten können verschie-

dene Teile des Bildes angezeigt werden. Sie können aber auch automatischen Bildlauf durchführen, indem Sie die linke Maustaste innerhalb des Client-Bereichs gedrückt halten und dann die Maus aus dem Client-Bereich bewegen.

Der Quelltext dazu sieht folgendermaßen aus (Sie finden ihn in der Datei SCROLAPP.CPP):

```
#include <owl.h>

// Deklaration von TScrollApp, Nachfahre von TApplication
class TScrollApp : public TApplication {
public:
  TScrollApp(LPSTR AName, HANDLE hInstance, HANDLE
             hPrevInstance, LPSTR lpCmdLine, int nCmdShow) :
             TApplication(AName, hInstance, hPrevInstance,
             lpCmdLine, nCmdShow) {};
  virtual void InitMainWindow();
};

// Deklaration von TScrollWindow, Nachfahre von TWindow
class TScrollWindow : public TWindow
{
public:
    TScrollWindow(LPSTR ATitle);
    virtual void Paint(HDC PaintDC, PAINTSTRUCT& PaintInfo);
};

/* Konstruktor für TScrollWindow, setzt Bildlaufleistenstile und
   initialisiert das Scroller-Objekt. */
TScrollWindow::TScrollWindow(LPSTR ATitle) : TWindow(NULL, ATitle)
{
  Attr.Style |= WS_VSCROLL | WS_HSCROLL;
  Scroller = new TScroller(this, 8, 15, 80, 60);
}

/* Antwortet auf eine empfangene "Zeichne"-Botschaft durch Neu-
   zeichnen von Rechtecken. Beachten Sie, daß die BeginView-Funktion
   des Scroller bereits aufgerufen wurde. Sie setzt den Ursprung
   des Grafikfensters relativ zur aktuellen Bildlaufposition. */

void TScrollWindow::Paint(HDC PaintDC,PAINTSTRUCT&)
{
  int X1, Y1, I;

  for (I = 0; I <= 49; ++I)
  {
    X1 = 10 + I*8;
    Y1 = 30 + I*5;
    Rectangle(PaintDC, X1, Y1, X1 + X1, X1 + Y1 * 2);
  }
}

// Initialisiert TScrollApp's MainWindow vom Typ TScrollWindow
```

```
void TScrollApp::InitMainWindow()
{
  MainWindow = new TScrollWindow("Boxes");
}

// Starten von ScrollApp
int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow)
{
  TScrollApp ScrollApp("ScrollApp", hInstance, hPrevInstance,
                       lpCmdLine, nCmdShow);
  ScrollApp.Run();
  return ScrollApp.Status;
}
```

Automatischer Bildlauf und Nachführung

Der automatische Bildlauf von *TScroller* ist standardmäßig aktiv. Sie können diese Eigenschaft ausschalten, wenn Sie das *AutoMode*-Datenelement in TScroller auf *FALSE* setzen. Das besitzende Fenster kann dies nach der Initialisierung des *TScroller*-Objekts in seinem Konstruktor durchführen:

```
Scroller = new TScroller(this, 8, 15, 80, 60);
Scroller->AutoMode = FALSE;
```

Nun kann der Bildlauf nur noch mit Hilfe der Bildlaufleisten durchgeführt werden.

Ein weiterer Vorteil des automatischen Bildlaufes liegt darin, daß sich der Fensterinhalt immer schneller verschiebt, je weiter Sie die Maus aus dem Client-Bereich des Fensters bewegen.

Neben dem automatischen Bildlauf führt das obige Beispielprogramm den Client-Bereich gemäß des Bildlaufs nach, wenn der Bildlauf vom Anwender durch Verschieben des Positionszeigers angefordert wird. Mit anderen Worten, das Bild wird entsprechend der Verschiebung des Positionszeigers verschoben. Durch diese unmittelbare Rückmeldung gelangt der Anwender zum gewünschten Teil des Bildes, ohne die Maustaste loslassen zu müssen.

In einigen Situationen ist Nachführung jedoch nicht wünschenswert. Wenn Sie beispielsweise eine umfangreiche Textdatei anzeigen, würde das Nachführen die Geschwindigkeit herabsetzen, da immer wieder von Platte gelesen und der gewünschte Textabschnitt für jede Bewegung des Positionszeigers angezeigt werden muß. In diesem Fall ist es günstiger, wenn Sie das Nachführen ausschalten:

```
Scroller->TrackMode = FALSE;
```

Während der Bewegung der Positionszeiger findet kein Bildlauf statt. Erst wenn der Anwender die Maustaste losläßt, wird der Bildlauf im Client-Bereich durchgeführt und der richtige Ausschnitt des Bildes angezeigt.

Die Schrittweiten und den Bildlaufbereich ändern

In den bisherigen Beispielen sind wir davon ausgegangen, daß die Schrittweiten und die Bereichswerte zum Zeitpunkt der *TScroller*-Initialisierung bekannt sind. In vielen Fällen ist diese Information aber nicht vorhanden oder kann sich ändern, etwa wenn sich die Größe der anzuzeigenden Informationen ändert. Dann müssen die Bereichswerte (und eventuell auch die Schrittweiten) neu gesetzt oder geändert werden. Kennen Sie bei der Initialisierung die Werte nicht, können Sie *NULL* an den *TScroller*-Konstruktor übergeben.

Die *SetRange*-Elementfunktion übernimmt zwei **long**-Argumente: die Anzahl der horizontalen und vertikalen Schrittweiten, die den gesamten Bildlaufbereich festlegen. *SetRange* sollte immer dann verwendet werden, wenn sich die Größe des Bildes ändert. Nehmen wir an, Sie möchten ein Bild mit einer Breite von 100 Einheiten und einer Höhe von 300 Einheiten anzeigen. Folgender Befehl setzt den dafür korrekten Bildlaufbereich:

```
Scroller->SetRange(100, 300);
```

Sind die Einheiten bei der Initialisierung des *TScroller*-Objekts nicht bekannt, müssen ihre Werte gesetzt werden, bevor verschoben werden kann. Diese Werte könnten beispielsweise in der *SetupWindow*-Elementfunktion des Fensters angegeben werden:

```
void ScrollWindow::SetupWindow()
{
  TWindow::SetupWindow();
  Scroller -> XUnit = 10;
  Scroller -> YUnit = 20;
}
```

Die Bildlaufposition ändern

Fenster können mit den *TScroller*-Elementfunktionen *ScrollTo* und *ScrollBy* zur Durchführung des Bildlaufs veranlaßt werden. Jede dieser Elementfunktionen hat zwei Argumente vom Typ **long** für die horizontalen und vertikalen Schrittweiten. Wenn es zum Beispiel erforderlich wird, die Bildlaufposition auf die obere linke Ecke des Bildes zu setzen, verwenden Sie *ScrollTo*:

```
Scroller->ScrollTo(0, 0);
```

Wenn ein Bild in der vertikalen Richtung 400 Schrittweiten lang ist, kann die Bildlaufposition folgendermaßen auf die Mitte des Bildes gesetzt werden:

```
Scroller->ScrollTo(0, 200);
```

Die Elementfunktion *ScrollBy* verschiebt die Bildlaufposition um eine bestimmte Anzahl von Schritten nach oben, unten, links oder rechts. Negative Werte bewegen die Bildlaufposition auf den Ursprungspunkt zu, also zur oberen

linken Ecke, positive Werte verschieben die Position nach rechts unten. Wenn Sie den Client-Bereich um 10 Schritte nach rechts und um 20 Schritte nach unten verschieben möchten, verwenden Sie folgenden Befehl:

```
Scroller->ScrollBy(10, 20);
```

Die Seitengröße bestimmen

Per Voreinstellung wird die Seitengröße (*XPage* und *YPage*) entsprechend der Client-Bereichsgröße des Fensters festgelegt. Der Scroller wird benachrichtigt, wenn sich die Größe seines besitzenden Fensters ändert. Die Elementfunktion *WMSize* des besitzenden Fensters ruft die *SetPageSize*-Elementfunktion ihres Scrollers auf, die dann die Datenelemente *XPage* und *YPage* besetzt. Die dabei verwendeten Werte basieren auf der Größe des Client-Bereichs und den Werten von *XUnit* und *YUnit*. Wenn Sie diesen Mechanismus überschreiben und die Seitengröße direkt angeben wollen, müssen Sie die von Ihrem Fensterobjekt geerbte *WMSize*-Elementfunktion ändern:

```
void TTestWindow::WMSize(RTMessage Msg)
{
  TWindow::WMSize(Msg);
}
```

Nun können Sie *XPage* und *YPage* direkt von Ihrem Fenster-Konstruktor aus festlegen (oder in einem von *TScroller* abgeleiteten Konstruktor):

```
TScrollWindow::TScrollWindow(PTWindowsObject AParent, LPSTR
                             ATitle) : TWindow(AParent, ATitle)
{
  Attr.Style |= WS_VSCROLL | WS_HSCROLL;
  Scroller = new TScroller(this, 8, 15, 80, 400);
  Scroller -> XPage = 40;
  Scroller -> YPage = 100;
}
```

Zeichenelementfunktionen für den Bildlauf optimieren

Die obige Beispielanwendung zeichnet 50 Rechtecke, versucht aber nicht festzustellen, ob alle Rechtecke auch im Client-Bereich des Fensters angezeigt werden können. So kann es vorkommen, daß während des Programmlaufs viel Zeit und Mühe auf das Zeichnen von nicht anzeigbaren Grafiken verwendet wird. Die Funktion *TScroller::IsVisibleRect* kann in einer *Paint*-Elementfunktion eingesetzt werden, um das Zeichnen in Fenstern zu optimieren.

Die folgende *ScrollWindow::Paint*-Elementfunktion legt mit *IsVisibleRect* fest, ob die Windows-Funktion *Rectangle* aufgerufen wird oder nicht. *Rectangle* nimmt Argumente in Geräteeinheiten entgegen, während *IsVisibleRect* mit

Schrittweiten arbeitet. Aus diesem Grund müssen die Werte *X1* und *Y1* (der Ursprung des Rechtecks) und (*X2-X1*) und (*Y2-Y1*) (Breite und Höhe des Rechtecks) vor dem Aufruf von *IsVisibleRect* durch die jeweiligen Schrittweiten dividiert werden:

```
void TScrollWindow::Paint(HDC PaintDC, PAINTSTRUCT& PaintInfo)
{
  int X1, Y1, X2, Y2, I;

  for (I= 0; I <= 49; ++I)
  {
    X1 = 10 + I*8;
    Y1 = 30 + I*5;
    X2 = X1 + X1;
    Y2 = X1 + Y1 * 2;
    if (Scroller->IsVisibleRect(X1/Scroller->XUnit,
                                Y1/Scroller->YUnit,
                                (X2 - X1)/Scroller->XUnit,
                                (Y2 - Y1)/Scroller->YUnit))
      Rectangle(PaintDC, X1, Y1, X2, Y2);
  }
}
```

Editier- und Dateifenster

ObjectWindows stellt zwei Nachfahren von *TWindow* zur Verfügung, die speziell zum Editieren von Text geeignet sind. Instanzen der Klasse *TEditWindow* sind Texteditier-Fenster, die aber keine Datei lesen oder in eine Datei schreiben können. Instanzen der Klasse *TFileWindow*, die von *TEditWindow* abgeleitet ist, definieren zusätzliche Verhaltensweisen, die für die Ausführung von Ein-/Ausgabeprozeduren erforderlich sind. Sie können Instanzen von diesen Klassen in Ihren eigenen Anwendungen einsetzen. Sie können auch spezialisierte Editoren erstellen, die von diesen Klassen abgeleitet sind. Programme mit Editier- oder Dateifenstern müssen die Datei editwnd.h bzw. filewnd.h und bestimmte ObjectWindows-Ressourcen einbinden. Lesen Sie zu diesem Punkt den Abschnitt »Aufbau einer ObjectWindows-Anwendung« in Kapitel 2 »Schritt für Schritt durch Windows«.

Editierfenster

Der Text in einem *TEditWindow*-Editierfenster steht tatsächlich in einem untergeordneten *TEdit*-Steuerelement und wird auch dort bearbeitet. Es ist also das *TEdit*-Steuerelement, das die Arbeit leistet und dafür von ObjectWindows mit weitreichenden Texteditierfähigkeiten versehen ist.

Der Konstruktor von *TEditWindow* speichert in seinem *Editor*-Datenelement die Referenz auf die Instanz von *TEdit*, die es erzeugt hat. Die Elementfunktion *WMSize*, die immer aufgerufen wird, wenn sich die Größe von *TEditWindow* ändert, bestimmt die Größe von *Editor* so, daß der Client-Bereich von *TEditWindow* ausgefüllt wird. Die Elementfunktion *WMSetFocus* übergibt den Fokus immer dann an *Editor*, wenn *TEditWindow* den Fokus erhält.

Mit *TEditWindow* kann das in *TEdit* vorhandene *Search*-Verhalten weiter ausgebaut werden. Es ist zuständig für den größten Teil der Suchen- und Ersetzen-Fähigkeiten, die dem Anwender eines *TEditWindow*-Fensters zur Verfügung stehen. *TEdit*-Steuerelemente wissen, wie der in ihnen enthaltene Text gesucht werden kann. Die Fähigkeit, Text zu ersetzen hat aber *TEditWindow*. *TEditWindow* hat weiterhin eine »Nächster«-Funktion, die durch die Pflege einer *TSearchStruct*-Struktur gewährleistet ist. So ist letztlich *TEditWindow* allein für die Suchen-/Ersetzen-Anwenderschnittstelle verantwortlich.

Das folgende Programm, EWNDTEST.CPP, verwendet ein Editierfenster, in dem der Anwender Text für eine einfache (nicht-funktionale) elektronische Mailbox-Anwendung ändern kann (siehe Abbildung 10.3).

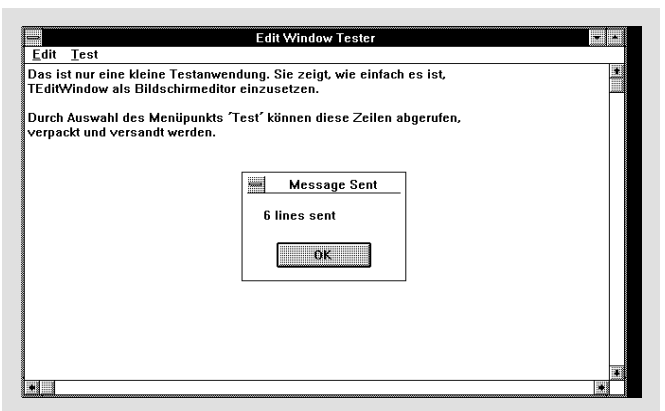


Abbildung 10.3
Ein Editierfenster

Dies ist die Datei EWNDTEST.CPP:

```
#include <owl.h>
#include <stdio.h>
#include <editwnd.h>
#include "ewndtest.h"

class TTestApp : public TApplication
{
public:
  TTestApp(LPSTR AName, HANDLE HInstance, HANDLE hPrevInstance,
           LPSTR lpCmdLine, int nCmdShow) : TApplication(AName,
           hInstance, hPrevInstance, lpCmdLine, nCmdShow) {}
  virtual void InitMainWindow();
  virtual void InitInstance();
};
```

```
class TTestWindow : public TEditWindow
{
public:
  TTestWindow(PTWindowsObject AParent, LPSTR ATitle);
  virtual void CMSendText(RTMessage Msg)
    = [CM_FIRST + CM_SENDTEXT];
};

TTestWindow::TTestWindow(PTWindowsObject AParent, LPSTR ATitle) :
                         TEditWindow(AParent, ATitle)
{
  Assign.Menu("COMMANDS");
}

void TTestWindow::CMSendText(RTMessage)
{
  int Lines;
  char Text[20];

  Lines = Editor->GetNumLines();
  sprintf(Text, "%d lines sent", Lines);
  MessageBox(HWindow, Text, "Message Sent", MB_OK);
}

void TTestApp::InitMainWindow()
{
  MainWindow = new TTestWindow(NULL, Name);
}

void TTestApp::InitInstance()
{
  TApplication::InitInstance();
  HAccTable = LoadAccelerators(hInstance, "EDITCOMMANDS");
}

int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow)
{
  TTestApp TestApp("Edit Window Tester" hInstance,
                   hPrevInstance, lpCmdLine, nCmdShow);
  TestApp.Run();
  return TestApp.Status;
}
```

Dies ist die Datei EWNDTEST.RC:

```
#include "windows.h"
#include "ewndtest.h"
#include "owlrc.h"

rcinclude stdwnds.dlg
rcinclude editacc.rc
```

```
COMMANDS MENU LOADONCALL MOVEABLE PURE DISCARDABLE
BEGIN
  POPUP "&Edit"
  BEGIN
    MenuItem  "&Undo\aAlt+BkSp", CM_EDITUNDO
    MenuItem  SEPARATOR
    MenuItem  "&Cut\aShift+Del", CM_EDITCUT
    MenuItem  "C&opy\aIns", CM_EDITCOPY
    MenuItem  "&Paste\aShift+Ins", CM_EDITPASTE
    MenuItem  "&Delete\aDel", CM_EDITDELETE
    MenuItem  "C&lear All\aCtrl+Del", CM_EDITCLEAR
  END
  POPUP "&Search"
  BEGIN
    MenuItem  "&Find...", CM_EDITFIND
    MenuItem  "&Replace...", CM_EDITREPLACE
    MenuItem  "&Next\aF3", CM_EDITFINDNEXT
  END
  MenuItem  "&Send Text", CM_SENDTEXT
END
```

Datei-fenster

Der Konstruktor von *TFileWindow* nimmt den Namen einer Datei als Argument entgegen. *TFileWindow::SetupWindow* baut das Dateifenster auf, indem die Inhalte dieser Datei in das Datenelement *Editor* gelesen werden, das von *TEditWindow* abgeleitete *TEdit*-Steuerelement. *TFileWindow* überschreibt *CanClose*, um den Anwender zum Sichern des aktuell editierten Textes aufzufordern, bevor das Dateifenster geschlossen wird.

TFileWindow verfügt über vier Elementfunktionen, mit deren Hilfe Anwenderaktionen zum Öffnen oder Speichern von Dateien beantwortet werden: *CMFileNew*, *CMFileOpen*, *CMFileSave* und *CMFileSaveAs*. Der Anwender fordert diese Aktionen durch Auswahl des entsprechenden Menüpunktes an. Dazu stehen folgende Bezeichner zur Verfügung:

Tabelle 10.4 Dateifenster-Elementfunktionen und Menü-IDs

Elementfunktion	Menü-ID für den Aufruf
CMFileNew	*CM_FILENEW*
CMFileOpen	*CM_FILEOPEN*
CMFileSave	*CM_FILESAVE*
CMFileSaveAs	*CM_FILESAVEAS*

Jede dieser Anforderungen, bei denen der Anwender einen Dateinamen angeben muß, verwendet eine Instanz von *TFileDialog*, mit deren Hilfe der Anwender Laufwerk, Verzeichnis und Dateinamen bestimmt.

Sie können Dateifenster ohne Modifikation als selbständige Texteditoren verwenden. Sicher wird es aber auch vorkommen, daß Sie zur Erhöhung der Funktionalität Ihre eigenen Klassen von *TFileWindow* ableiten wollen. So könnten Sie beispielsweise eine Klasse von *TFileWindow* ableiten, die die Schrift des im Editor angezeigten Textes ändert. Normalerweise werden Sie diese Option dem Anwender als Drop-Down-Liste in einem von Ihnen erzeugten Menü anbieten. Wenn Sie die von *TFileWindow* zur Verfügung gestellte Anwenderschnittstelle für die Dateiein-/ausgabe verwenden wollen, müssen Sie Menüeinträge mit den in der obigen Tabelle aufgeführten Bezeichnern erstellen.

11. Dialog-objekte

Ein Dialog ist ein Schnittstellenelement, dessen Erstellungsattribute eine Windows-Ressourcendatei angibt. Die Erstellungsattribute in der Definition der Dialogressoruce definieren die Darstellungsform und die Positionierung des Dialogs und seiner Steuerelemente.

Dialoge sind normalerweise untergeordnete Fenster, die spezielle eingabespezifische Aufgaben ausführen. So kann ein untergeordneter Dialog Anwendereingaben aufnehmen, die der Konfiguration eines Druckers dienen.

ObjectWindows unterstützt drei von *TDialog* abgeleitete Klassen, die häufig benötigte Eingaben vom Anwender abrufen:

TInputDialog ruft eine einzeilige Texteingabe ab;

TFileDialog ruft einen Dateinamen ab;

TSearchDialog ruft einen Text zum Suchen und Ersetzen in einem Editierfenster ab.

Dialogressourcen verwenden

Für ein Dialog-Schnittstellenelement muß eine Ressourcendefinition in einer Windows-Ressourcendatei existieren. Die Ressourcendefinition, die einen eindeutigen Bezeichner hat, beschreibt das Aussehen und die Positionierung des Dialogelements und seiner Steuerelemente. Die Ressourcendefinition legt aber nicht das Verhalten und die Eigenschaften des Dialogs fest.

In einer ObjectWindows-Anwendung definiert ein *TDialog*-Objekt das Verhalten und die Eigenschaften eines Dialogs. Sein Konstruktor übernimmt den Bezeichner der Ressourcendefinition als Parameter. Diese Ressourcendefinition wird verwendet, wenn ein Schnittstellenelement für die Verknüpfung mit *TDialog* erstellt wird.

In Kapitel 2 »Schritt für Schritt durch Windows« finden Sie im Abschnitt »Aufbau einer ObjectWindows-Anwendung: Vorbereitungen« Erläuterungen zu diesem Thema. Sie müssen unbedingt darauf achten, die Ressourcen der compilierten Anwendung hinzuzufügen.

Untergeordnete Dialogobjekte verwenden

Die Verwaltung eines untergeordneten Dialogobjekts ähnelt der Verwaltung von Pop-up-Fensterobjekten. Dialoge sind untergeordnete Fenster eines übergeordneten Fensters. Dialoge und Pop-up-Fenster werden während der Lebensdauer ihrer übergeordneten Fenster mehrmals initialisiert, angezeigt und freigegeben. Dialoge und Pop-up-Fenster, die nur für kurze Zeit angezeigt werden, können mit Hilfe einer einzigen Elementfunktion initialisiert, angezeigt und freigegeben werden, ohne dabei als Datenelemente ihrer übergeordneten Fenster gespeichert zu werden. Diejenigen Dialoge aber, die für einen längeren Zeitraum erstellt werden, werden als Datenelemente gespeichert.

Dialoge erzeugen

Ein *TDialog*-Konstruktor hat zwei Parameter: eine Referenz auf das übergeordnete Fensterobjekt und den Bezeichner der Dialogressource. *TDialog* überschreibt seinen Konstruktor, so daß für den Ressourcenbezeichner entweder ein Wert vom Typ *LPSTR* oder vom Typ *int* übergeben werden kann:

```
ADialog = new TSampleDialog(this, "SampleDialog");
```

oder

```
ADialog = new TSampleDialog(this, 101);
```

Es empfiehlt sich, für die Bezeichner Konstanten zu verwenden:

```
#define ID_SAMPLERESOURCE 101;
.
.
.
ADialog = new TSampleDialog(this, ID_SAMPLERESOURCE);
```

Dialogobjekte sollten immer auf dem Heap (nicht dem Stack) angelegt werden. ObjectWindows löscht das Dialogobjekt automatisch, wenn das entsprechende Windows-Element geschlossen wird.

Dialoge erstellen und ausführen

Die Verknüpfung zwischen einem untergeordneten Dialogobjekt und einem Schnittstellenelement wird bei der *Ausführung* oder *Erstellung* von *TDialog* hergestellt.

Ein untergeordneter Dialog wird *ausgeführt*, wenn der weitere Ablauf der Anwendung von einer Anwendereingabe abhängt. Die Ausführung stoppt, bis die Eingabe erfolgt ist und der Dialog geschlossen wird. So führt beispielsweise

ein Fenster zum Editieren von Text, das geschlossen wird, einen Dialog aus, um festzustellen, ob die noch nicht gespeicherten Änderungen gespeichert werden sollen. Der weitere Verlauf (speichern oder nicht speichern) hängt von der Anwendereingabe ab. Die Ausführung stoppt (das Fenster wird nicht geschlossen), bis die Eingabe erfolgt ist. Während der Ausführung eines Dialogs ist das übergeordnete Fenster deaktiviert und nicht in der Lage, auf Anwendereingaben zu antworten.

Ein untergeordneter Dialog wird *erstellt*, wenn die Anwendereingabe Auswirkungen auf den Verlauf der Ausführung hat, aber die Ausführung auch ohne Eingaben weiterläuft. Ein Fenster zum Editieren von Text könnte z.B. einen Suchen-Dialog erstellen, der bei der ersten Aktivierung der Suche durch den Anwender die Eingabe eines Such-Strings verlangt. Wenn dieser Dialog vom Anwender nicht explizit geschlossen wird, kann er für die Eingabe neuer Such-Strings für aufeinanderfolgende Suchvorgänge verwendet werden. Dabei wird die normale Ausführung (die Textbearbeitung) fortgeführt. Ein erstellter Dialog deaktiviert sein übergeordnetes Fenster nicht.

Das Ausführen eines Dialogs unterscheidet sich vom Erstellen eines Dialogs noch in einem weiteren Punkt: Wenn Sie einen Dialog ausführen, liefert der Dialog einen Wert zurück, der die vom Anwender ausgewählte Option repräsentiert. Einfache Dialoge zeigen häufig Fragen, die der Anwender beantworten muß und verfügen zu diesem Zweck über *OK*- und *Cancel*-Schalter. Wenn der Anwender einen der beiden Schalter anklickt, wird der Dialog beendet. Er liefert den Bezeichner des angeklickten Schalters als Ergebnis der Ausführung zurück.

Windows läßt nur einen Integerwert als Rückgabewert eines ausgeführten Dialogs zu. Wenn komplexere Eingaben ermöglicht werden sollen, können Sie den in ObjectWindows zur Verfügung stehenden Datentransfermechanismus anpassen oder einen selbstdefinierten Transferpuffer Ihrer abgeleiteten Dialogklasse ausfüllen.

Informationen über den Datentransfer finden Sie in Kapitel 12 »Steuerelementobjekte« im Abschnitt »Steuerelement-Daten übertragen«.

Modale und nicht-modale Dialoge

Ein Dialog, der ausgeführt ist, wird *modaler* Dialog genannt; ein nicht-modaler Dialog ist ein Dialog, der erstellt wird. *TDialog* kann sowohl mit einem modalen als auch mit einem nicht-modalen Dialog durch den Aufruf seiner Elementfunktionen *Execute* und *Create* verknüpft werden. Diese Elementfunktionen sollten jedoch nicht direkt aufgerufen werden. Rufen Sie zur sicheren *Ausführung* eines modalen Dialogs die Elementfunktion *ExecDialog* des Anwendungsobjekts auf und zur *Erstellung* eines nicht-modalen Dialogs die Elementfunktion *MakeWindow*. *ExecDialog* und *MakeWindow* werden von *TModule* abgeleitet.

```
// Ausführen eines modalen Dialogs
if (GetModule()->ExecDialog(new TSampleDialog(this,
                            "SampleDialog")) == IDOK)
// weitere Aktionen

// Erstellen eines nicht-modalen Dialogs
GetModule()->MakeWindow(new TSampleDialog(this, "SampleDialog"));
```

Einen untergeordneten Dialog beenden

Zum Schließen eines modalen oder nicht-modalen Dialogs rufen Sie seine *CloseWindow*-Elementfunktion auf (die *Ok*-Botschaftsantwortfunktion führt diesen Aufruf aus). *CloseWindow* schließt den Dialog erst, wenn ein Aufruf von *CanClose* ergeben hat, daß dies zulässig ist. Bei einem modalen Dialog überträgt *CloseWindow* standardmäßig die Daten des Dialogs. Soll ein Dialog in jedem Fall (d.h. ohne Berücksichtigung des Ergebnisses von *CanClose*) geschlossen werden, rufen Sie *ShutDownWindow* auf (die *Cancel*-Botschaftsantwortfunktion führt diesen Aufruf aus).

Normalerweise werden Sie *ShutDownWindow* nicht überschreiben. Allerdings kann ein Überschreiben von *CloseWindow* erforderlich sein, damit ein nicht-modaler Dialog *TransferData* aufruft. Sie können *CanClose* auch überschreiben, um vor dem Schließen des Dialogs die vom Anwender eingegebenen Daten auf ihre Gültigkeit zu überprüfen (wenn die Eingabe unzulässig ist, sollten Sie ein Meldungsfenster anzeigen oder einen Signalton erzeugen).

Bei einem modalen Dialog wird der an *CloseWindow* oder *ShutDownWindow* übergebene Wert zum Rückgabewert des *Execute*-Aufrufs und wird auch von *TModule::ExecDialog* zurückgeliefert. Wird kein Wert angegeben, ist *IDCANCEL* der Standard-Rückgabewert. Da nicht-modale Dialoge keine Rückgabewerte haben, muß für diese Dialoge auch kein Wert übergeben werden.

Hinweis: Lesen Sie zu diesem Punkt auch den Abschnitt »Steuerelement-Daten übertragen« in Kapitel 12 »Steuerelementobjekte«.

Einen Dialog als Hauptfenster verwenden

Sie können nicht-modale Dialoge als Hauptfenster Ihrer Anwendung einsetzen. Der Grund dafür könnte sein, daß Ihr Hauptfenster viele Steuerelemente enthält. Bei der Verwendung eines Dialogs als Hauptfenster können Sie mit einem Dialogressourcen-Editor die Erstellungsattribute und die Steuerelemente des Fensters auf bequeme Art und Weise definieren. Sie können eine von *TDialog* abgeleitete Klasse genauso als Hauptfenster verwenden wie eine von *TWindow* abgeleitete Klasse. Die Erstellung erfolgt in *InitMainWindow*. Das Programm CALC.CPP im EXAMPLES-Verzeichnis verwendet einen nicht-modalen Kalkulationsdialog als Hauptfenster.

Sie können einen nicht-modalen Dialog auch dann als Hauptfenster der Anwendung einsetzen, wenn Sie die Definition des Schnittstellenelements von der Anwendung trennen wollen. Auf diese Weise kann das Aussehen des Hauptfensters geändert werden, ohne daß die Anwendung geändert oder erneut compiliert werden muß.

Ein Taschenrechnerprogramm kann beispielsweise einen nicht-modalen Dialog als Hauptfenster haben, in dem die Tasten des Rechners als Schalter in einer Dialogressource festgelegt sind. Dadurch wird es möglich, daß das Hauptfenster zusätzlich über ein Menü, ein Symbol und einen Mauszeiger verfügen kann.

Sie werden auf einige Schwierigkeiten treffen, wenn Sie Tastenkürzel in einer

Anwendung einsetzen wollen, die als Hauptfenster einen Dialog verwendet. Windows unterstützt den Gebrauch von Tastenkürzeln in nicht-modalen Dialogen nicht vollständig. Es gibt dennoch »Tricks«, die Ihnen dabei helfen, diese Einschränkung zu umgehen. Das Programm CALC.CPP verwendet einen dieser Tricks.

Eine Windows-Fensterklasse für einen nicht-modalen Dialog definieren

Wir haben Ihnen bereits das Konzept der Registrierungsattribute für ein Fenster und den Definitionsprozeß dieser Attribute vorgestellt (siehe den Abschnitt »Windows-Fensterklassen-Registrierung« in Kapitel 10 »Fensterobjekte«). Windows benötigt die Registrierung bestimmter Attribute eines Fensters, bevor es erzeugt werden kann. Zu diesen Registrierungsattributen gehören das Symbol und der Mauszeiger. ObjectWindows registriert voreingestellte Attribute für Ihre Fenster. Wenn Sie die Voreinstellungen ändern möchten, müssen Sie die Elementfunktionen GetWindowClass und GetClassName überschreiben.

Windows stellt vordefinierte Registrierungsattribute für Ihre Dialoge bereit, die Sie für einen modalen Dialog nicht ändern können: Windows erlaubt dies nicht. Für einen nicht-modalen Dialog ist eine Änderung dieser Attribute jedoch möglich. Dazu definieren Sie für Ihr Fenster die Elementfunktionen *GetWindowClass* und *GetClassName* nach Ihren Wünschen neu. Außerdem müssen Sie den Namen der Windows-Fensterklasse Ihres Dialogs in der Ressourcendefinition des Dialogs bestimmen. Dabei handelt es sich um den Namen, den die Elementfunktion *GetClassName* des Dialogs zurückliefert.

Wenn Sie in der Ressourcendefinition des Dialogs keinen Namen festlegen, verfügt der Dialog über die von Windows definierten Standard-Registrierungsattribute. Geben Sie einen Windows-Fensterklassennamen an, kann der Dialog nicht als modaler Dialog ausgeführt werden.

Steuerelemente manipulieren und Botschaften bearbeiten

Windows unterstützt die Erstellung und Verwaltung von Steuerelementen durch ein übergeordnetes Fenster nur in geringem Umfang. Fenster- und Steuerelementobjekte verwalten diese Aufgaben in einer ObjectWindows-Anwendung (vgl. Kapitel 12 »Steuerelementobjekte«. Dort finden Sie Informationen über die Verwendung von Steuerelementen in einem Fenster).

Windows verwaltet die Erstellung von Steuerelementen in einem Dialog und unterstützt in gewissem Umfang deren Verwaltung. Aus diesem Grund sind in ObjectWindows die Steuerelemente eines Dialogs nicht per Voreinstellung mit den Steuerelementobjekten verknüpft. Sie können diese Verknüpfung jedoch herbeiführen, um die Kommunikation zwischen Dialog und Steuerelement zu erleichtern.

Es gibt eine zweigleisige Kommunikation zwischen einem Dialogobjekt und seinen Steuerelementen. In der einen Richtung muß der Dialog seine Steuerelemente manipulieren und abfragen, zum Beispiel eine Liste ausfüllen oder Text aus einem Editierfeld abrufen. In der anderen Richtung müssen die Steuerele-

mente eines Dialogs den Dialog benachrichtigen, wenn ein den Dialog betreffendes Ereignis eingetreten ist. So benachrichtigt beispielsweise ein Schalter seinen übergeordneten Dialog davon, daß er gedrückt wurde.

Dialog-Steuerelemente manipulieren

Windows definiert Steuerelementbotschaften und Standardantworten für jeden Steuerelementtyp. So besteht die Standardantwort einer Liste auf eine *LB_GETCURSEL*-Botschaft im Zurückliefern des Index des ausgewählten Eintrags. Ein Dialog kommuniziert mit seinen Steuerelementen durch Senden von Steuerelementbotschaften, in denen ein Steuerelementbezeichner, ein Botschaftsbezeichner und botschaftsspezifische Parameter bestimmt sind.

In ObjectWindows senden Sie eine Botschaft zum Steuerelement eines Dialogs, indem Sie die Elementfunktion *SendDlgItemMsg* aufrufen, die folgende vier Parameter erwartet:

- den Bezeichner des Steuerelements,
- den Bezeichner der Botschaft,
- einen Parameter von Typ WORD, der die Botschaftsspezifikationen (die zugehörigen Daten) enthält,
- einen Parameter vom Typ LONG, der weitere Spezifikationen aufnimmt.

Im folgenden Programmbeispiel wird die Elementfunktion *SendDlgItemMsg* von *TestDialog* aufgerufen, um "Item 1" in das Steuerelement der Liste einzufügen (durch Senden einer *LB_ADDSTRING*-Botschaft).

```
void TTestDialog::FillListBox()
{
  SendDlgItemMsg(ID_LISTBOX, LB_ADDSTRING, NULL,
                 (DWORD) "Item 1");
}
```

Sie können die Kommunikation zwischen Dialogobjekten und ihren Steuerelementen erleichtern, indem Sie die Steuerelemente mit Steuerelementobjekten verknüpfen, die Elemente der Dialogklasse sind. Denken Sie daran, daß die Steuerelemente eines Dialogs nicht per Voreinstellung mit den ObjectWindows-Steuerelementobjekten verknüpft sind. Anschließend können Sie (gemäß den Grundsätzen des objektorientierten Programmierens) die Steuerelemente mit Hilfe der Datenelemente und Elementfunktionen des Objekts manipulieren, anstatt Botschaften über die Windows-API an das Schnittstellenelement des Steuerelements zu senden.

Zu diesem Zweck erzeugen Sie ein Steuerelementobjekt im Konstruktor Ihres Dialogs. Verwenden Sie den Konstruktor des Steuerelementobjekts, der nur das übergeordnete Objekt und den Ressourcenbezeichner als Parameter entgegennimmt. Die Erstellungsattribute müssen nicht übergeben werden, da Windows das Element bereits erstellt hat. Nun können Sie die Steuerelemente des Dialogs so manipulieren und abfragen wie die Steuerelemente eines Fensters. Im folgenden sehen Sie die damit zusammenhängenden Änderungen der Deklaration des Konstruktors und der Elementfunktion *FillListBox* von *TTestDialog*:

```
class TTestDialog : public TDialog
{
public:
  ...
  PTListBox ListBox;
  TTestDialog(PTWindowsObject AParent, LPSTR AName);
  ...
};

TTestDialog::TTestDialog(PTWindowsObject AParent, LPSTR AName) :
                         TDialog(AParent, AName)
{
  ListBox = new TListBox(this, ID_LISTBOX);
}
void TTestDialog::FillListBox()
{
  ListBox->AddString("Item 1");
}
```

Benachrichtigungsbotschaften der Steuerelemente beantworten

Ein Steuerelement sendet eine Benachrichtigungsbotschaft an sein übergeordnetes Fenster, wenn ein dieses Fenster betreffendes Ereignis eintritt. In ObjectWindows bestimmt das Dialogobjekt eine Antwort auf Botschaften, die ein bestimmtes Steuerelement sendet, indem eine auf der ID des untergeordneten Fensters basierende Botschaftsantwortfunktion definiert wird. Diese wird aufgerufen, wenn der Dialog eine Benachrichtigungsbotschaft von dem angegebenen untergeordneten Fenster empfängt.

In der folgenden Beispieldeklaration deklariert *TTestDialog* auf der ID des untergeordneten Fensters basierende Botschaftsantwortfunktionen für zwei seiner Steuerelemente:

```
class TTestDialog : public TDialog {
  ...
void HandleButtonMsg(RTMessage Msg) =
  [ID_FIRST + ID_BUTTON];
void HandleListBoxMsg(RTMessage Msg) =
  [ID_FIRST + ID_LISTBOX];
};
```

Jede Benachrichtigungsbotschaft des Steuerelements verfügt über einen Code, eine Integer-Konstante, die die betreffende Aktion bezeichnet. Die Auswahl eines Listeneintrags produziert zum Beispiel eine Benachrichtigungsbotschaft mit dem Code *LBN_SELCHANGE*. Das Anklicken eines Schalters produziert eine Botschaft mit dem Code *BN_CLICKED*. Der Benachrichtigungscode wird in *Msg.LP.Hi*, dem höherwertigen WORD von *Msg.LParam* übergeben.

Die Antworten der Botschaftsantwortfunktionen, die auf der ID des untergeordneten Fensters basieren, hängen normalerweise von dem in *Msg.LP.Hi* übergebenen Benachrichtigungscode ab. Sehen Sie sich folgendes Beispiel an:

```
void TTestDialog::HandleListBoxMsg(RTMessage Msg) {
  switch (Msg.LP.Hi)
  {
    case LBN_SELCHANGE :
      // Auswahländerung
      break;
    case LBN_DBLCLK :
      // Auswahl durch Doppel-Klick
      break;
    ...
  }
}
```

Sie haben nun gesehen, wie Antworten auf Benachrichtigungsbotschaften von Steuerelementen für Ihre Dialogobjekte definiert werden. Aber eventuell möchten Sie, daß die Steuerelemente Ihres Dialogs auf eigene Benachrichtigungsbotschaften antworten. So könnten Sie beispielsweise einen Schalter in Ihrem Dialog haben, der eine Aktion ausführen soll, wenn er gedrückt wird. Zu diesem Zweck müssen Sie eine benachrichtigungsspezifische Elementfunktion in einer von Ihnen definierten Steuerelementklasse erstellen. Dann verknüpfen Sie ein ObjectWindows-Steuerelementobjekt mit dem Steuerelement Ihres Dialogs (vgl. dazu den vorhergehenden Abschnitt).

Beispiel für die Kommunikation zwischen Dialog und Steuerelement

Im folgenden finden Sie die von *TDialog* abgeleitete Klasse *TTestDialog*. *TTestDialog* definiert *HandleButtonMsg* und *HandleListBoxMsg*, die auf Benachrichtigungsbotschaften antworten, die vom Schalter und von der Liste an den Dialog gesendet werden. *HandleButtonMsg* wird aufgerufen, wenn der Schalter gedrückt wird, und füllt die Liste mit Texteinträgen aus. Diese Einträge werden durch Aufrufe der Elementfunktion *SendDlgItemMsg* zur Verfügung gestellt. *TTestDialog* wird in der Elementfunktion *ExecDialog* des übergeordneten Fensters als modaler Dialog ausgeführt. In Abbildung 11.1 sehen Sie das Dialogfenster.

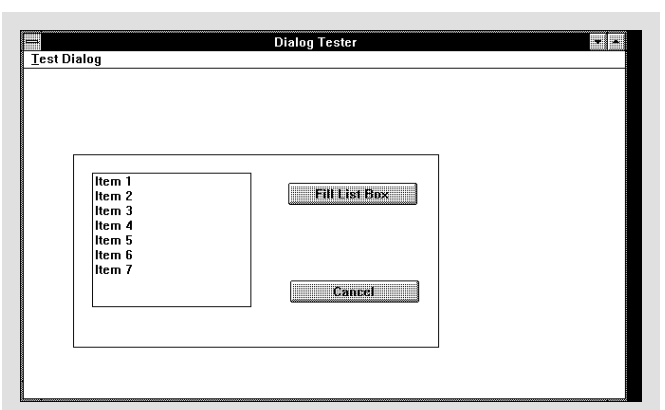


Abbildung 11.1
Eine Dialog-Anwendung

Das vollständige Programm finden Sie unter dem Namen DIALTEST.CPP im OWL\EXAMPLES-Verzeichnis .

Erweitertes Beispiel für einen Dialog

Das Programm VDLGAPP.CPP definiert einen komplexen Dialog, der den Anwender zur Eingabe von Text auffordert und dann den Text auf Zulässigkeit überprüft. Es zeigt ein Dialogfenster an, das folgende Editierfelder enthält: Name des Beschäftigten, Sozialversicherungsnummer und ID des Beschäftigten.

Wenn der Anwender *OK* anklickt, überprüft die *CanClose*-Elementfunktion, ob die drei Werte korrekt eingegeben wurden. Erst dann kann der Dialog geschlossen werden. In Abbildung 11.2 sehen Sie diese Anwendung.

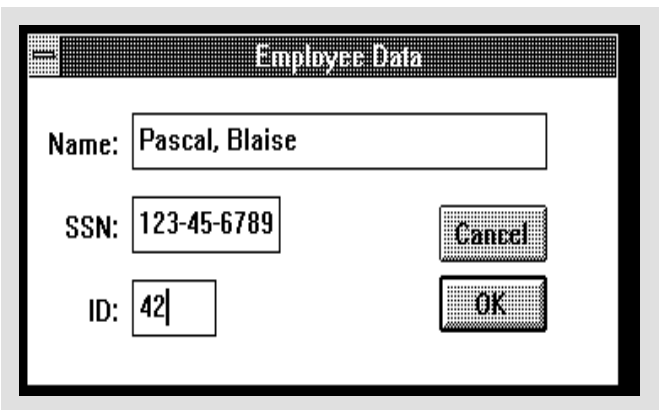


Abbildung 11.2 Ein Dialog, der die Anwendereingabe auf Korrektheit überprüft

Das vollständige Programm finden Sie unter dem Namen VDLGAPP.CPP im OWL\EXAMPLES-Verzeichnis.

Eingabedialoge

Eingabedialoge sind spezielle, in der ObjectWindows-Bibliothek definierte Dialoge. Eingabedialoge zeigen eine Aufforderung an und fordern eine einzeilige Texteingabe durch den Anwender. Dies ist eine in Windows-Anwendungen häufig auszuführende Aufgabe. Zur Verwendung von Eingabedialogen muß die Datei inputdia.h in Ihr Programm und die Datei INPUTDIA.DLG in die .RC-Datei des Programms eingebunden werden.

Der Konstruktor eines Eingabedialogs nimmt fünf Parameter entgegen:

- einen Zeiger auf das übergeordnete Fensterobjekt,
- einen Zeiger auf den als Dialogtitel angezeigten Text,

- einen Zeiger auf den als Eingabeaufforderung angezeigten Text,
- einen Zeiger auf einen Textpuffer, der mit dem vom Anwender eingegebenen Text gefüllt wird,
- die Größe des Texteingabepuffers.

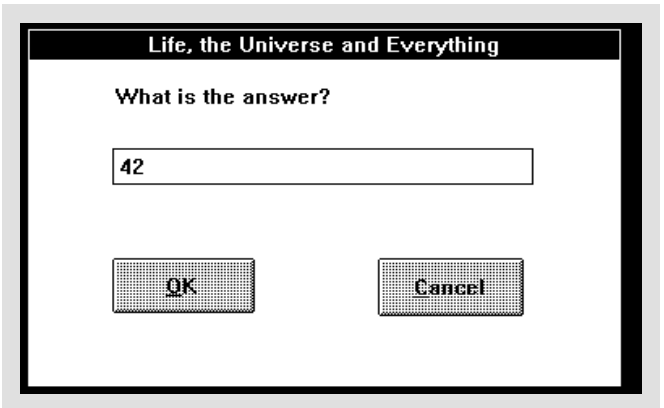


Abbildung 11.3
Ein Eingabedialog

Im folgenden Beispiel wird ein Eingabedialog erstellt. *EditText* wird mit dem vom Anwender in das Editierfeld des Dialogs eingegebenen Text ausgefüllt, sobald der *OK*-Schalter des Eingabedialogs gedrückt wird.

```
PTInputDialog AnInputDialog;
char EditText[79];

strcpy(EditText, "");
AnInputDialog = new TInputDialog(this, "Caption", "Prompt",
                                 EditText, sizeof(EditText));
```

Der Textpuffer, der an den Konstruktor des Eingabedialogs übergeben wird, sollte einen Vorgabe-String enthalten, der bei der ersten Anzeige des Eingabedialogs erscheint. Achten Sie darauf, daß der Textpuffer zumindest mit einem Null-String besetzt ist.

Die Zeiger auf die Texteinträge des Eingabedialogs müssen für die gesamte Lebensdauer des Dialogs gültig sein. Daran müssen Sie vor allem dann denken, wenn Sie einen Eingabedialog als nicht-modalen Dialog ausführen. In den meisten Fällen werden Eingabedialoge jedoch als modale Dialoge ausgeführt, wie es auch im folgenden Beispiel demonstriert wird:

```
void TSampleWindow::GetName()
{
  char EditText[79];
  strcpy(EditText, "Frank Borland");
  if (ExecDialog(new TInputDialog(this, "Data Entry",
      "Enter Name", EditText, sizeof(EditText))) == IDOK)
    MessageBox(HWindow, EditText, "Name is:", MB_OK);
  else MessageBox(HWindow, EditText, "Name is still:", MB_OK);
}
```

Datei-Dialoge

Datei-Dialoge sind ein weiterer Typ von vordefinierten Dialogen, die von ObjectWindows in der Klasse *TFileDialog* zur Verfügung gestellt werden. Verwenden Sie einen Datei-Dialog immer dann, wenn Sie vom Anwender einen Dateinamen erfahren wollen, wie in den Anwendungsfunktionen *File Open*, *Save* und *Save As*. Sehen Sie sich dazu Abbildung 11.4 an.

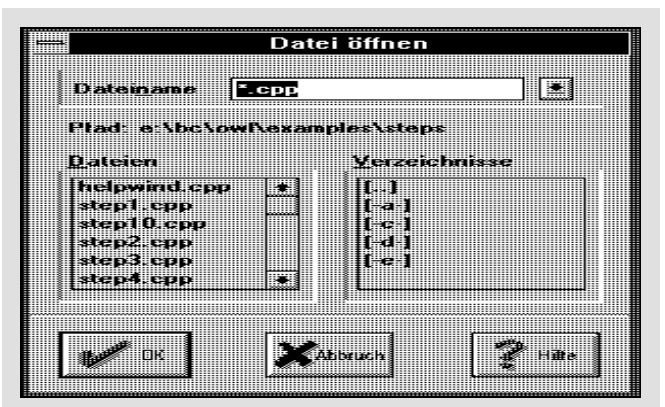


Abbildung 11.4
Ein Datei-Dialogfenster

Zur Verwendung von *TFileDialog* muß in Ihr Programm die Datei filedial.h und in die .RC-Datei des Programms die Datei filedial.dlg eingebunden werden.

Der Konstruktor eines Datei-Dialogfensters hat drei Parameter: einen Zeiger auf das übergeordnete Fenster, einen Ressourcenbezeichner und einen Dateinamen oder eine Maske (abhängig davon, ob das Datei-Dialogfenster für das Öffnen oder Speichern einer Datei verwendet wird). Der übergebene Ressourcenbezeichner ist entweder der Bezeichner für das Standard-Datei-Dialogfenster zum Öffnen oder für das Speichern der Datei (*SD_FILEOPEN* bzw. *SD_FILESAVE*). Für ein Datei-Dialogfenster zum Öffnen von Dateien wird im Dateinamensparameter eine Dateinamensmaske übergeben; für ein Datei-Dialogfenster zum Speichern von Dateien enthält dieser Parameter einen voreingestellten Dateinamen. Der Dateinamensparameter wird auch als Rückgabe-Puffer für den Dateinamen verwendet, der vom Anwender in beiden Dialogfensterarten ausgewählt werden kann.

Im folgenden Beispiel wird ein Datei-Dialogfenster zum Öffnen von Dateien ausgeführt. "*.*" wird als Dateimaske an den Konstruktor des Datei-Dialogs übergeben. Dadurch werden alle Dateien des aktuellen Verzeichnisses in der Dateinamensliste angezeigt, wenn das Fenster zum ersten Mal angezeigt wird:

```
char TempName[MAXPATH];
.
.
.
_fstrcpy(TempName, "*.*");
```

```
if (GetApplication()->ExecDialog(new TFileDialog(this,
                                  SD_FILEOPEN, TempName)) == IDOK))
// Öffnen der in TempName angegebenen Datei
...
```

Im folgenden Beispiel wird ein Datei-Dialog zum Speichern einer Datei ausgeführt. Als voreingestellter Dateiname wird ein Null-String an den Konstruktor des Datei-Dialogs übergeben. Dadurch wird kein voreingestellter Name ausgewählt, wenn das Datei-Dialogfenster zum ersten Mal angezeigt wird:

```
char TempName[MAXPATH];
...
TempName[0] = '\0';
if (GetApplication()->ExecDialog(new TFileDialog(this,
                          SD_FILESAVE), TempName)) == IDOK)
// Speichern in die in TempName angegebene Datei
...
```

12. Steuerelement-objekte

Benutzerschnittstellen-Elemente, die den Transfer von Anwendereingaben erleichtern, werden *Steuerelemente* genannt. So handelt es sich z.B. bei *Auswahlfeldern* und *Listen*, die den Anwendern von Windows-Anwendungen vertraut sind, um zwei Arten von Steuerelementen. Für Windows sind Steuerelemente einfach spezialisierte Fenster. In ObjectWindows dient *TWindow* als Basisklasse für die Klasse *TControl*.

Von *TControl* abgeleitete Klassen stellen die »Repräsentanten« für die Windows-Steuerelemente dar. In Ihren Anwendungen werden Sie Instanzen dieser abgeleiteten Klassen verwenden. Sehen Sie sich dazu die folgende Tabelle an:

Tabelle 12.1 Von ObjectWindows unterstützte Windows-Steuerelemente

Element	Klasse	Verwendung
Liste	*TListBox*	Eine bildlauffähige Liste mit Einträgen, wie z.B. Dateien, aus denen ausgewählt werden kann.
Bildlaufleiste	*TScrollBar*	Eine selbständige Bildlaufleiste, ähnlich den Bildlaufleisten in bildlauffähigen Fenstern und Listen.
Aktionsschalter	*TButton*	Ein Aktionsschalter mit zugehörigem Text.
Auswahlfeld	*TCheckBox*	Ein Auswahlfeld mit zugehörigem Text.
Schaltfeld	*TRadioButton*	Ein Schaltfeld, das ein- oder ausgeschaltet sein kann; es wird normalerweise in Gruppen mit mehreren Schaltfeldern verwendet, die sich gegenseitig ausschließ en.
Gruppe	*TGroupBox*	Ein der Gruppierung anderer Steuerelemente dienendes statisches Rechteck mit Text in der oberen linken Ecke.
Editierfeld	*TEdit*	Ein Feld, in das Text eingegeben werden kann.
Statisches Steuerelement	*TStatic*	Ein nur zur Ausgabe dienendes Textfeld.
Kombinationsfenster	*TComboBox*	Eine Kombination aus Liste und Editierfeld.

Abbildung 12.1 zeigt die Bildschirmdarstellung einiger typischer Steuerelemente.

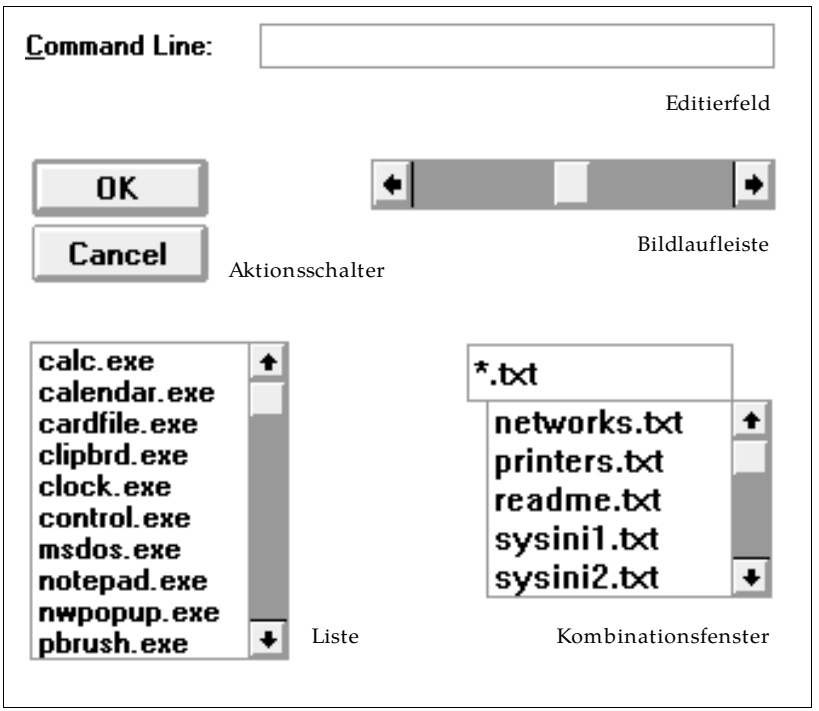


Abbildung 12.1 Beispiele für Steuerelemente

ObjectWindows definiert Klassen für alle in der vorhergehenden Tabelle aufgeführten Steuerelementtypen, die als untergeordnete Fenster innerhalb anderer Fenster verwendet werden. Steuerelemente, die zu Dialogfenstern gehören, sind Elemente der Benutzerschnittstelle. Sie sind in einer Dialogressource definiert und benötigen daher keine korrespondierenden ObjectWindows-Objekte (vgl. dazu Kapitel 11 »Dialogobjekte«). In diesem Kapitel wird die Funktionsweise der Steuerelementobjekte skizziert. Auß erdem wird erläutert, wie sie innerhalb eines Fensters verwendet werden können.

Steuerelementobjekte verwenden

Die Lebensdauer eines Steuerelements entspricht der Lebensdauer seines übergeordneten Fensters. Steuerelemente werden normalerweise zusammen mit ihrem übergeordneten Fenster erstellt und freigegeben. Dieses Verhalten wird von ObjectWindows unterstützt. Jedes bei einem übergeordneten Fenster aufgelistete untergeordnete Fenster wird bei der Erstellung des übergeordneten Fensters angelegt und zusammen mit diesem freigegeben.

Steuerelementobjekte initialisieren und erstellen

Sie werden in den meisten Fällen Ihre Steuerelementobjekte in den Konstruktoren der übergeordneten Fenster erzeugen. Dabei ist es aber nicht erforderlich, Referenzen auf die untergeordneten Fenster, die als Datenelemente der übergeordneten Fenster erstellt sind, zu verwalten. Sie können die Referenz eines untergeordneten Fensters jederzeit mit einem Aufruf der*ChildWithId*-Elementfunktion des übergeordneten Fensters abrufen (vorausgesetzt, daß Sie jedem untergeordneten Fenster im *AnID*-Parameter einen eindeutigen Bezeichner für einen Konstruktor aller von *TControl* abgeleiteten Klassen zugeordnet haben). Greift Ihr Programm häufig auf ein untergeordnetes Fenster zu, ist es einfacher, einen Zeiger als Datenelement seines übergeordneten Fensters zu speichern.

Die Konstruktoren der Steuerelementobjekte haben mindestens sechs Parameter: einen Zeiger auf das übergeordnete Fensterobjekt, die ID des Steuerelements, die x- und y-Koordinaten der linken oberen Ecke (bezüglich des Client-Bereichs des übergeordneten Fensters) und die Höhe und Breite. Diese Parameter werden für das Setzen der Erstellungsattribute des Steuerelementobjekts herangezogen, die in der Struktur *Attr* gespeichert werden. Zu den vom *TControl*-Konstruktor in *Attr.Style* gesetzten Stilen gehören *WS_CHILD*, *WS_VISIBLE*, *WS_TABSTOP* und *WS_GROUP*. Diese Voreinstellung kann in den abgeleiteten Klassen geändert werden, wie dies bei einigen der ObjectWindows-Steuerelementklassen geschehen ist. In Tabelle 10.1 in Kapitel 10, »Fensterobjekte«, sind die Fensterattribut-Datenelemente aufgeführt.

Create (bzw. *GetApplication*()->*MakeWindow*) sollte nicht für jedes Steuerelement direkt aufgerufen werden. Sie werden automatisch angelegt, wenn die Elementfunktion *SetupWindow* von *TWindowsObject* für das übergeordnete Fenster aufgerufen wird. Sie können *SetupWindow* in der abgeleiteten Klasse Ihres übergeordneten Fensters so überschreiben, daß der Aufbauprozeß für die Steuerelemente damit durchgeführt wird. Wenn Sie *SetupWindow* überschreiben, müssen Sie sicherstellen, daß die *SetupWindow*-Elementfunktion der Basisklasse für das Anlegen der Steuerelemente aufgerufen wird.

Wiederholend zusammengefaßt läßt sich sagen, daß Sie Datenelemente für Ihre übergeordneten Fenster definieren können, in denen die Zeiger auf ihre Steuerelemente gespeichert sind. Erstellen Sie danach die Steuerelementobjekte im Konstruktor des übergeordneten Fensters. Im letzten Arbeitsschritt definieren Sie *SetupWindow*, damit es die Steuerelemente (falls vorhanden) initialisiert. ObjectWindows kümmert sich um den Rest.

Steuerelementobjekte freigeben und entfernen

In ObjectWindows liegt die Verantwortung für die Freigabe und das Entfernen von Steuerelementen beim übergeordneten Fenster. Ein Steuerelement wird immer freigegeben und das zugehörige Steuerelementobjekt entfernt, wenn das übergeordnete Fenster freigegeben und entfernt wird.

Steuerelemente und Botschaftsbearbeitung

Die Kommunikation zwischen einem Fensterobjekt und seinen Steuerelementobjekten ähnelt weitgehend der Kommunikation zwischen einem Dialogobjekt und seinen Steuerelementen. In beiden Fällen manipuliert und überwacht das übergeordnete Fenster seine Steuerelemente. Die Steuerelemente ihrerseits senden Benachrichtigungsbotschaften von Steuerelementereignissen an das übergeordnete Fenster. Das übergeordnete Fenster kann für diese Benachrichtigungsbotschaften Elementfunktionen definieren, die auf der ID des untergeordneten Fensters basieren. Per Voreinstellung sind die Steuerelemente eines Dialogs jedoch nicht mit ObjectWindows-Steuerelementobjekten verknüpft. Steuerelemente eines Fensters sind dagegen immer mit Steuerelementobjekten verknüpft.

In diesem Abschnitt wird die Kommunikation zwischen Fenster und Steuerelement, in Kapitel 11 »Dialogobjekte« die Kommunikation zwischen Dialog und Steuerelement erläutert.

Steuerelemente eines Fensters manipulieren

Fenster manipulieren und überwachen ihre Steuerelemente, indem sie entsprechende Elementfunktionen bereitstellen.

Haben die Steuerelementobjekte eines Fensters entsprechende Datenelemente im Fensterobjekt des übergeordneten Fensters, können die Steuerelement-Elementfunktionen ganz einfach aufgerufen werden:

```
ListBox->AddString("Scotts Valley");
```

Wird kein Zeiger auf das Steuerelementobjekt in einem Datenelement bereitgestellt, muß das Steuerelement aus der Liste der untergeordneten Fenster durch den Aufruf der Elementfunktion *ChildWithID* des übergeordneten Fensters ermittelt werden:

```
PTListBox TheListBox;
...
TheListBox = (PTListBox)ChildWithID(ID_LISTBOX);
TheListBox->AddString("Scotts Valley");
```

Benachrichtigungsbotschaften der Steuerelemente beantworten

Ereignisse in den Steuerelementen eines Fensters erzeugen Benachrichtigungsbotschaften (vgl. dazu Kapitel 6 »Fenster anzeigen«, Abschnitt »Steuerelement-Ereignisse beantworten«). In den meisten Fällen antworten Fensterobjekte auf Benachrichtigungsbotschaften von Steuerelementen durch den Aufruf von Botschaftsantwortfunktionen, die auf der ID des untergeordneten Fensters basieren. Sehen Sie sich dazu das folgende Beispiel an:

```
class TSampleWindow : public TWindow {
public:
```

```
  PTListBox ListBox;
  PTButton Button1, Button2;

  virtual void HandleListBoxMsg(RTMessage Msg) =
    [ID_FIRST + ID_LISTBOX];
  virtual void HandleButton1Msg(RTMessage Msg) =
    [ID_FIRST + ID_BUTTON1];
  virtual void HandleButton2Msg(RTMessage Msg) =
    [ID_FIRST + ID_BUTTON2];
}
```

In Ihren Botschaftsantwortfunktionen enthält *Msg.LP.Hi* den Steuerelement-Benachrichtigungscode, zum Beispiel *LBN_SELCHANGE* oder *BN_CLICKED*. Schreiben Sie eine Antwort, die die wichtigen Benachrichtigungscodes verwaltet:

```
void TSampleWindow::HandleListBoxMsg(RTMessage Msg)
{
  switch (Msg.LP.Hi) {
  case LBN_SELCHANGE:
    // Auswahländerung
    break;

  case LBN_DBLCLK:
    // Auswahl durch Doppel-Klick
    break;
  }
}
```

Es kann Situationen geben, in denen ein Steuerelementobjekt selbst auf eine Benachrichtigungsbotschaft antworten soll. Bauen Sie hierzu ein derartiges Verhalten in das Steuerelement ein. So könnten Sie beispielsweise einen Schalter entwerfen, der seinen Text ändert, wenn er vom Anwender angeklickt wird oder ein Editierfeld anlegen, das nur die Eingabe von Ziffern zuläßt. In diesen speziellen Fällen ist es effizienter, wenn Sie eine Steuerelement-Klasse mit den gewünschten Eigenschaften anlegen, anstatt die Steuerelemente für jedes übergeordnete Fenster zu wiederholen. Das Ergebnis wird eine selbständige Klasse sein, die in vielen Anwendungen verwendet werden kann.

Um ein Steuerelementobjekt so zu programmieren, daß es seine Benachrichtigungsbotschaften direkt beantwortet, definieren Sie eine Botschaftsantwortfunktion für Benachrichtigungen in einer von *TControl* abgeleiteten Klasse:

```
class TSpecializedListBox : public TListBox {
public:
  virtual void LBNSelChange(RTMessage Msg) =
    [NF_FIRST + LBN_SELCHANGE];
  virtual void LBNDblClick(RTMessage Msg) =
    [NF_FIRST + LBN_DBLCLK];
}
```

Sie müssen dabei die Summe von *NF_FIRST* und dem Steuerelementbenachrichtigungscode in der Erweiterung der Funktionsdeklaration für Elementfunk-

tionen verwenden, die auf Benachrichtigungen antworten. Benachrichtigungen sind Steuerelementbotschaften, die Windows sendet, wenn das entsprechende Steuerelement vom Anwender »bedient« wird.

Die Deklaration für *TSpecializedListBox::LBNSelChange* lautet:

```
void TSpecializedListBox::LBNSelChange(RTMessage Msg)
{
  // Auswahländerung
}
```

Soll auch das übergeordnete Fenster auf eine Benachrichtigung antworten, muß die Botschaftsantwortfunktion, die die entsprechende Benachrichtigung beantwortet, die Elementfunktion *DefNotifyProc* des Steuerelements aufrufen.

Steuerelement-Fokus und die Tastatur

Haben Sie jemals in einem Dialogfenster den Fokus über die Tastatur von Steuerelement zu Steuerelement übergeben? Um von Steuerelement zu Steuerelement (in der Reihenfolge ihrer Erstellung) zu springen, kann die Tab-Taste verwendet werden. Für das Verschieben des Fokus zwischen den Schaltfeldern in einer Gruppe stehen auß erdem die Pfeiltasten zur Verfügung. In ObjectWindows kann diese Tastaturschnittstelle für die Steuerelemente in Fenstern emuliert werden. Dazu rufen Sie die Elementfunktion *EnableKBHandler* im Fenster-Konstruktor auf.

Listen

Eine Liste ist die einfachste Art, den Anwender eines Windows-Programms zur Auswahl eines Listeneintrags aufzufordern. Abhängig von der Art des Programms können Sie den Anwender beispielsweise auffordern, aus einer Liste von Dateien, Druckern, Formen oder Früchten auszuwählen. Listen sind in der Klasse *TListBox* gekapselt. *TListBox* definiert vier Elementfunktionen zur Ausführung folgender Aufgaben: Listen erstellen, Listeneinträge ändern, Listeneinträge abfragen und vom Anwender ausgewählten Eintrag feststellen.

Listen initialisieren und erstellen

Einer der Konstruktoren von *TListBox* nimmt das übergeordnete Fenster, die ID und die Steuerelementmaß e *X*, *Y*, *W* und *H* als Argumente entgegen.

Das folgende Beispiel könnte im Konstruktor eines übergeordneten Fensters stehen:

```
ListBox = new TListBox(this, ID_LISTBOX, 20, 20, 340, 100);
```

Der *TListBox*-Konstruktor ruft den Konstruktor von *TControl* auf, und fügt dann *LBS_STANDARD* zu den voreingestellten Stilen hinzu, die in *Attr.Style* für die Liste festgelegt sind. *LBS_STANDARD* ist eine Kombination von Stilen, die

- die Einträge in der Liste in alphabetischer Reihenfolge anlegen (*LBS_SORT*),
- das übergeordnete Fenster benachrichtigen, wenn ein Ereignis in der Liste eintritt (*LBS_NOTIFY*),
- für die Liste einen Rahmen anlegen (*WS_BORDER*),
- für die Liste eine vertikale Bildlaufleiste festlegen (*WS_VSCROLL*).

Sie können *Attr.Style*, das die voreingestellten Stile enthält, nach der Initialisierung der Liste ändern. Sollen beispielsweise die Einträge in einer Liste nicht alphabetisch sortiert werden, schreiben Sie folgende Programmzeilen:

```
ListBox = new TListBox(this, ID_LISTBOX, 20, 20, 340, 100);
ListBox->Attr.Style &= ~LBS_SORT;
```

Sie können *Attr.Style* auch im Konstruktor einer von *TListBox* abgeleiteten Klasse ändern.

Wie bei allen untergeordneten Fenstern werden auch die visuellen Elemente der Listen vom übergeordneten Fenster erstellt.

Listen ändern

Nach der Erstellung einer Liste müssen Sie diese mit Einträgen auffüllen. Die Einträge müssen Strings oder besitzergezeichnet sein. Im weiteren Verlauf der Arbeit werden Sie bei Bedarf Einträge hinzufügen, entfernen oder die gesamte Liste löschen. Zum Eintragen und Hinzufügen von Einträgen rufen Sie die Elementfunktion *AddString* der Liste auf:

```
ListBox->AddString("Item 1");
ListBox->AddString("Item 2");
ListBox->AddString("Item 3");
ListBox->AddString("Item 4");
ListBox->AddString("Item 5");
ListBox->AddString("Item 6");
```

Ist *AddString* nicht erfolgreich, liefert sie einen negativen Wert zurück. Eine Liste verwaltet Strings – ähnlich wie ein String-Array – mit Hilfe eines Index. Wenn *ListBox* ein Array wäre, würde 'Item 1' an Position *ListBox*[*0*] gespeichert, 'Item 2' an Position *ListBox*[*1*], 'Item 3' an Position *ListBox*[*2*], etc. Bei jedem Aufruf von *AddString* wird der angegebene String per Voreinstellung in alphabetischer Reihenfolge oder am Ende der Liste eingefügt, wenn der Stil *LBS_SORT* nicht beinhaltet.

Unabhängig davon, ob die Liste sortiert ist, können Sie ein neues Element mit *InsertString* an einer bestimmten Indexposition einfügen:

```
ListBox->InsertString("Item 1.5", 1);
```

Dieser Befehl verschiebt die Strings an den Indexpositionen, die größer oder gleich 1 sind, um eine Stelle. Die obigen sieben Elementfunktionsaufrufe ergeben folgende Liste:

Index	Eintrag
0	"Item 1"
1	"Item 1.5"
2	"Item 2"
3	"Item 3"
4	"Item 4"
5	"Item 5"
6	"Item 6"

Sehen Sie sich Abbildung 12.2 an.

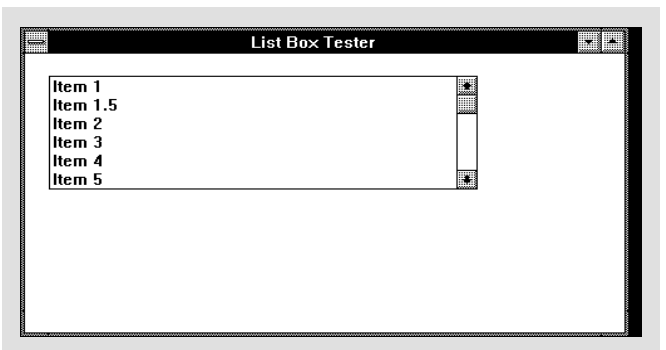


Abbildung 12.2 Eine ausgefüllte Liste

InsertString hängt den String an das Listenende, wenn die übergebene Indexposition -1 ist. Mit *DeleteString* können Sie einen Eintrag entfernen. Der folgende Elementfunktionsaufruf löscht den String an der Position 1 ("Item 1.5") und vermindert den Index der folgenden Strings um 1:

```
ListBox->DeleteString(1);
```

ClearList löscht alle Strings in der Liste:

```
ListBox->ClearList();
```

Listen abfragen

Mit Hilfe von sechs Elementfunktionen können Sie sich Informationen über die in einem Listenobjekt enthaltene Liste verschaffen.

GetCount liefert die Anzahl der Einträge in der Liste.

FindString und *FindExactString* durchsuchen die Liste ab der angegebenen Indexposition nach einem String. Dabei wird (falls nötig) wieder zum Beginn der Liste

gesprungen und bis zur Indexposition weitergesucht. *FindString* sucht einen String in der Liste, der die angegebene Zeichenfolge enthält, *FindExactString* sucht nach einem String, der mit der angegebenen Zeichenfolge beginnt.

GetString ermittelt den String in der Liste, der an der durch das Integer-Argument angegebenen Indexposition steht. Ein *LPSTR*-Argument zeigt auf einen Puffer, der den String aufnehmen soll. *GetString* liefert auch einen Integerwert zurück, der die Länge des Strings angibt.

GetStringLen liefert die Länge des Strings an der angegebenen Position zurück, den String selbst jedoch nicht.

GetSelIndex liefert die Indexposition des momentan gewählten Strings (kann nur für Listen mit einem Eintrag eingesetzt werden).

Aus einer Liste wählen

Es gibt drei Dinge, die der Anwender mit einer Liste tun kann: Er kann durch die Liste blättern und einen Eintrag einmal oder zweimal anklicken. Wenn in einer Liste eine Anwenderaktion ausgeführt wird, sendet Windows eine *Listen-Benachrichtigungsbotschaft* an das übergeordnete Fenster der Liste.

Jede Listen-Benachrichtigungsbotschaft (*LBN*) enthält einen Listen-Benachrichtigungscode (eine Integerkonstante) in *Msg.LP.Hi*, die die Art der Aktion angibt. In der folgenden Tabelle finden Sie eine Beschreibung der gebräuchlichsten *LBN_*-Codes:

Tabelle 12.2 Listen-Benachrichtigungsbotschaften

WParam	Aktion
LBN_SELCHANGE	Es wurde eine Option mit einem einzigen Maus-Klick gewählt.
LBN_DBLCLK	Es wurde eine Option mit einem doppelten Maus-Klick gewählt.
LBN_SETFOCUS	Der Anwender hat der Liste durch einmaliges oder doppeltes Anklicken eines Eintrags oder durch die Verwendung der Tab-Taste den Fokus übergeben. *LBN_SETFOCUS* geht den Botschaften *LBN_SELCHANGE* und *LBN_DBLCLK* voran.

Das übergeordnete Fenster beantwortet Benachrichtigungen einer Liste mit Hilfe einer Botschaftsantwortfunktion. Im folgenden sehen Sie die Botschaftsantwortfunktion eines übergeordneten Fensters, die die Benachrichtigungsbotschaften beantwortet:

```
void TSampleWindow::HandleListBoxMsg(RTMessage Msg)
{
  int Idx;
  char Str[10];

  if (Msg.LP.Hi == LBN_SELCHANGE)
  {
    Idx = ListBox->GetSelIndex();
```

```
      if (ListBox->GetStringLen(Idx) <= sizeof(Str))
      {
        ListBox->GetSelString(Str, sizeof(Str));
        MessageBox(HWindow, Str, "You selected:", MB_OK);
      }
    }
  }
```

Hinweis: Dieses Programmfragment stammt aus der Datei LBOXTEST.CPP

Der Anwender hat eine Auswahl getroffen, wenn das höherwertige *WORD* von *LParam* den Benachrichtigungscode *LBN_SELCHANGE* enthält. Ist dies der Fall, wird die Länge des ausgewählten Strings festgestellt, überprüft, ob der String in den *Str*-Puffer paß t, der String ermittelt und in einem Meldungsfenster angezeigt (vgl. Abbildung 12.3).

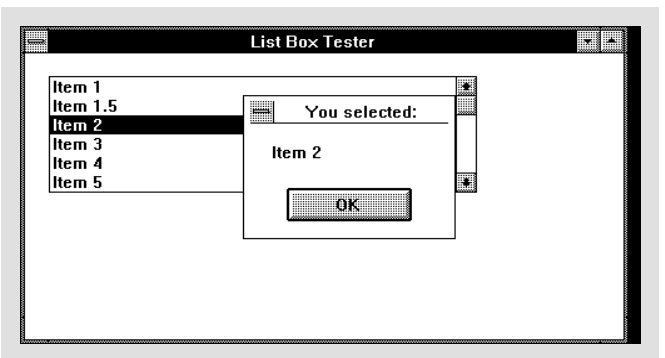


Abbildung 12.3 Antwort auf die Anwender-Auswahl eines Listeneintrags

Ausgewählte Listeneinträge werden mit vier *TListBox*-Elementfunktionen bearbeitet: *GetSelString*, *GetSelIndex*, *SetSelString* und *SetSelIndex*. Die *Get*-Elementfunktionen ermitteln den String und die Indexposition des ausgewählten Strings. Die *Set*-Elementfunktionen übergehen den Anwender und bewirken durch Bestimmung des Strings oder der Indexposition die Auswahl eines bestimmten Eintrags. Dadurch wird der String auch in den sichtbaren Bereich gebracht, wenn dies nicht bereits der Fall ist.

Das Programm *LBoxTest* erstellt ein Fenster mit einer Liste. Beim Start der Anwendung wird eine Liste im Hauptfenster angezeigt. Wenn der Anwender einen Listeneintrag auswählt, erscheint ein Dialogfenster, in dem der Listeneintrag angezeigt wird. Das Hauptfenster speichert eine Referenz auf das Steuerelement der Liste in seinem Datenelement *ListBox*. Dadurch wird die Manipulation des Steuerelements vereinfacht. Die vollständige Datei LBOXTEST.CPP finden Sie im Unterverzeichnis OWL\EXAMPLES.

Kombinationsfenster

Ein Kombinationsfenster ist eine Kombination aus zwei anderen Steuerelementen, einer Liste und einem Editierfeld. Es dient zum Abruf zweier unter-

schiedlicher Anwendereingaben: der Auswahl aus einer Liste oder der Eingabe von Text in Editierfelder.

TComboBox ist von der Klasse *TListBox* abgeleitet und erbt deren Elementfunktionen für das Ändern, Abfragen und Auswählen von Listeneinträgen. *TComboBox* verfügt über zusätzliche Elementfunktionen zur Manipulation des Listenteils im Kombinationsfenster, der in bestimmten Fällen auf Anforderung »ausklappt« (*Drop-down*-Liste).

Drei Arten von Kombinationsfenstern

Es gibt drei Arten von Kombinationsfenstern: einfache, ausklappende und ausklappende Listen-Kombinationsfenster. Abbildung 12.4 zeigt diese drei Arten und eine Liste.

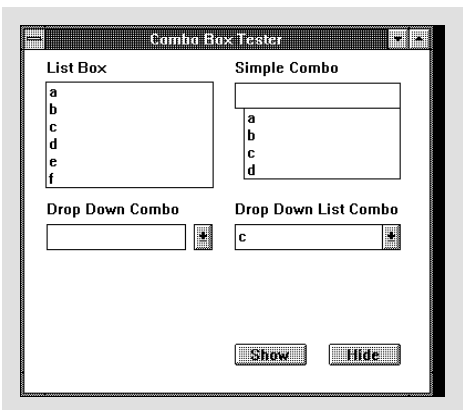


Abbildung 12.4 Die drei Arten von Kombinationsfenstern und eine Liste

Einfache Kombinationsfenster

Alle Kombinationsfenster zeigen immer ihren Editierbereich an, manche Kombinationsfenster können jedoch den Listenbereich ein- oder ausblenden. Einfache Kombinationsfenster können dies nicht, der Listenbereich wird immer angezeigt. Ein einfaches Kombinationsfenster enthält ein Editierfeld, in dem der Anwender Text eingeben und editieren kann. Entspricht der eingegebene Text einem Eintrag in der Liste, wird dieser Eintrag ausgewählt.

Ausklappende Kombinationsfenster

Ein ausklappendes Kombinationsfenster entspricht einem einfachen Kombinationsfenster mit einer einzigen Ausnahme: Zu Beginn ist der Listenbereich nicht eingeblendet. Dieser Bereich wird erst dann angezeigt, wenn der Anwender den Pfeil nach unten auf der rechten Seite des Editierfeldes anklickt. Die Liste wird nach einer Auswahl ausgeblendet. Ausklappende Kombinationsfenster sind sehr nützlich, wenn viele Steuerelemente in einen kleinen Bereich passen sollen. Wenn die ausklappenden Listen nicht benötigt werden, belegen sie nur soviel Platz wie ein Editierfeld.

Ausklappende Listen-Kombinationsfenster

Der Listenbereich eines derartigen Kombinationsfensters verhält sich wie der Listenbereich in einem ausklappenden Kombinationsfenster – er wird nur dann eingeblendet, wenn er benötigt wird, und danach wieder ausgeblendet. Die beiden ausklappenden Arten unterscheiden sich im Verhalten ihrer Editierfelder. Während sich die Editierbereiche der ausklappenden Kombinationsfenster wie reguläre Editierfelder verhalten, kann der Editierbereich in ausklappenden Listen-Kombinationsfenstern nur den Text genau eines Listeneintrags anzeigen. Die Zeichen dieses Textes können vom Anwender nicht geändert werden.

Einen Kombinationsfenstertyp auswählen

Ausklappende Listen-Kombinationsfenster sind besonders für Fälle geeignet, in denen nur Einträge ausgewählt werden können, die in der Liste stehen. So können zum Beispiel bei der Auswahl von Druckern nur diejenigen ausgewählt werden, auf die Ihr System zugreifen kann (siehe Abbildung 12.5)

Ausklappende Kombinationsfenster können jedoch auch Einträge akzeptieren, die nicht in der Liste aufgeführt sind. Eine Einsatzmöglichkeit für diese Fenster ist die Auswahl von Dateien, die geöffnet oder gespeichert werden sollen. Der Anwender kann sowohl in der Liste die Verzeichnisse nach der gewünschten Datei durchsuchen, als auch den vollen Pfad- und Dateinamen im Editierbereich eingeben, unabhängig davon, ob der Dateiname im Listenbereich angezeigt wird.

Kombinationsfenster erzeugen

Der Konstruktor von *TComboBox* benötigt die üblichen Parameter: das übergeordnete Fenster, die Steuerelement-ID, Position und Größe und zusätzlich die Parameter *AStyle* und *ATextLen* (siehe Abbildung 12.5).

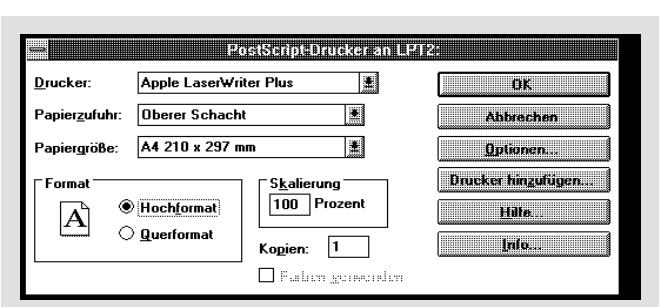


Abbildung 12.5 Ein aufspringendes Listen-Kombinationsfenster

Dem Konstruktor des Kombinationsfensters wird ein Standardstil (entweder *CBS_SIMPLE*, *CBS_DROPDOWN* oder *CBS_DROPDOWNLIST*) als Stil-Parameter übergeben. Das Datenelement *Attr.Style* des Kombinationsfensters wird mit

diesem Parameter und weiteren Standardstilen gesetzt. Sehen Sie sich dazu die folgende Programmzeile an:

```
Attr.Style = WS_CHILD | WS_VISIBLE | WS_GROUP | WS_TABSTOP |
             CBS_SORT | CBS_AUTOHSCROLL | WS_VSCROLL | AStyle;
```

Im folgenden Beispiel wird das Datenelement *Attr.Style* eines Kombinationsfensters so geändert, daß Windows die Liste im Kombinationsfenster nicht sortiert:

```
Combo3 = new TComboBox(this, ID_COMBO3, 190, 160, 150, 100
                       CBS_DROPDOWNLIST, 40);
Combo3->Attr.Style &= ~CBS_SORT;
```

Der Parameter *ATextLen* gibt die maximale Länge des Editierfeldes im Kombinationsfenster an.

Kombinationsfenster ändern

TComboBox definiert zwei Elementfunktionen, die den Listenbereich von ausklappenden Kombinationsfenstern und Listen-Kombinationsfenstern ein- und ausblenden: *ShowList* und *HideList*. Beide Elementfunktionen benötigen keine Argumente. Sie müssen diese Funktionen zum Ein- und Ausblenden der Liste nicht explizit aufrufen, wenn der Anwender den Pfeil nach unten rechts neben dem Editierbereich anklickt. Dieser Mechanismus ist in Kombinationsfenstern immer aktiviert. Diese Elementfunktionen eignen sich dazu, das Ein- und Ausblenden der Liste zu erzwingen.

Beispielprogramm: CBoxTest

Das Programm *CBoxTest* produziert die in Abbildung 12.4 gezeigte Anwendung und verwendet alle drei Arten von Kombinationsfenstern. *Combo*1 ist ein einfaches, *Combo2* ein ausklappendes Kombinationsfenster und *Combo3* ein ausklappendes Listen-Kombinationsfenster.

Das Anklicken der Schalter *Show* und *Hide* hat das Ein- und Ausblenden der Liste des rechten unteren Kombinationfensters *Combo3* zur Folge. Zu diesem Zweck werden die Elementfunktionen *ShowList* und *HideList* aufgerufen.

Das vollständige Programm finden Sie unter dem Namen CBOXTEST.CPP im Unterverzeichnis OWL\EXAMPLES.

Statische Steuerelemente

Statische Steuerelemente sind normalerweise unveränderbare Texteinheiten oder einfache Grafiken, die in einem Fenster oder Dialog auf dem Bildschirm

angezeigt werden. Der Anwender kommuniziert nicht mit diesen Elementen, obwohl das Programm deren Text als Ergebnis einer Anwenderaktion ändern kann. In Abbildung 12.6 sehen Sie verschiedene statische Elemente und ihre entsprechenden Windows-Stilkonstanten.

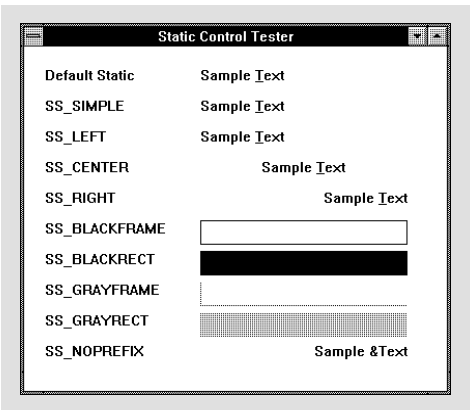


Abbildung 12.6
Statische Steuerelemente

Statische Steuerelemente erzeugen

Da statische Steuerelemente keine Benachrichtigungsbotschaften senden und normalerweise nicht vom Programm über ihre IDs manipuliert werden, müssen diese Elemente keine eindeutige ID haben. Per Konvention lautet die ID für statische Steuerelemente -1.

Zusätzlich zu den üblichen Konstruktor-Parametern für Steuerelemente hat der *TStatic*-Konstruktor einen weiteren Parameter, der die maximale Länge des Textes festlegt, der von *TStatic*-Objekten übertragen wird. Lesen Sie zu diesem Thema den Abschnitt »Steuerelement-Daten übertragen« am Ende dieses Kapitels. Dort finden Sie eine Erläuterung des Transfers von Daten und der Rolle des *TStatic*-Datenelements *TextLen*.

Die Konstruktoren von *TStatic* rufen den *TControl*-Konstruktor auf und fügen dann *SS_LEFT* (für Linksausrichtung des Texts) zum Datenelement *Attr.Style* hinzu und entfernen den Stil WS_TABSTOP daraus. Zur Änderung der voreingestellten Stile modifizieren Sie *Attr.Style*:

```
AStatic = new TStatic(this, -1, "&Text", 20, 50, 200, 24, 6);
AStatic->Attr.Style = AStatic->Attr.Style & ~SS_LEFT | SS_CENTER;
```

Wenn Sie die Stile für die statischen Steuerelemente nicht ändern wollen, müssen Sie auch keine Referenz in Form einer lokalen Variable mitführen. Wenn Sie – wie es normalerweise der Fall ist – diese Elemente nach ihrer Erstellung nicht manipulieren müssen, ist auch keine Referenz auf sie im Datenelement des übergeordneten Fensters erforderlich.

Eine für statische Elemente verfügbare Option ist das Unterstreichen eines oder mehrerer Zeichen im Textstring. Die Implementierung und ihr Ergebnis ähneln der Unterstreichung des ersten Zeichens einer Menüauswahl: Fügen Sie

ein &-Zeichen unmittelbar vor dem Zeichen ein, das unterstrichen werden soll. Wenn Sie zum Beispiel das T in 'Text' unterstreichen wollen, übergeben Sie den String '&Text' im Konstruktor-Aufruf.

Soll das &-Zeichen als solches in einem String verwendet werden, sollten Sie den statischen Windows-Stil *SS_NOPREFIX* einsetzen (vgl. Abbildung 12.6).

Informationen über statische Steuerelemente

Mit der *GetText*-Elementfunktion können Sie sich über den Text informieren, der in einem statischen Steuerelement steht.

Statische Steuerelemente ändern

TStatic verfügt über zwei Elementfunktionen, mit deren Hilfe Sie den Text eines statischen Steuerelements ändern können. *SetText* setzt den Text auf denjenigen, der mit dem *LPSTR*-Argument übergeben wird. *Clear* löscht den Text eines statischen Steuerelements. Der Text in Elementen, die mit dem Stil *SS_SIMPLE* erstellt wurden, kann jedoch nicht geändert werden.

Beispiel: Die Anwendung StatTest

Das Programm StatTest erstellt die Test-Anwendung für statische Elemente, die Sie in Abbildung 12.6 sehen. Beachten Sie, daß Beschriftungen wie *"Default Static"* und *"SS_SIMPLE"* ebenso statische Steuerelemente sind wie *"Sample Text"* und die schwarzen und grauen Felder.

Die Programmdatei finden Sie unter dem Namen STATTEST.CPP im Unterverzeichnis OWL\EXAMPLES.

Editierfelder

Editierfelder sind rechteckige Steuerelemente (mit oder ohne Rahmen) auf dem Bildschirm, die vom Anwender eingegebenen Text aufnehmen. Ein Editierfeld kann von einem Anwender mit Text ausgefüllt werden, der Text kann aber auch vom Programm geändert werden. Viele Operationen, die mit einem Editierfeld ausgeführt werden können (wie z.B. *Cut*, *Copy* und *Paste*), senden und erhalten Text zu bzw. aus der Zwischenablage. Editierfelder unterstützen folgende Operationen:

- Anwender-Texteingabe;
- Dynamische Textänderung durch die Anwendung;

- Kopieren und Einfügen in die und aus der Zwischenablage;
- Mehrzeiliges Editieren (hilfreich für Texteditoren).

Windows verfügt über Editierfelder, die auf viele Anwenderereignisse antworten können. Wenn Sie z.B. ein Wort in einem Editierfeld zweimal anklicken, ist das ganze Wort ausgewählt; wenn Sie Entf drücken, wird das Zeichen an der Mauszeiger-Position gelöscht. ObjectWindows erweitert die Fähigkeiten eines Windows-Editierfelds, indem es zusätzliche leistungsstarke Editierfunktionen zur Verfügung stellt. Folgende Abbildung zeigt ein Fenster mit Editierfeldern.

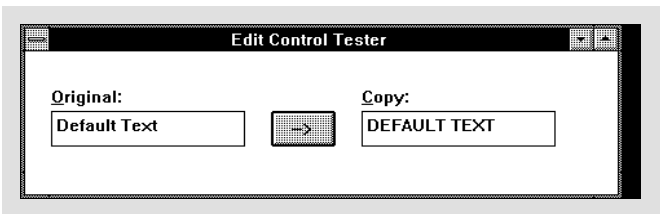


Abbildung 12.7
Ein Fenster mit Editierfeldern

Editierfelder erzeugen

Der Konstruktor von *TEdit* akzeptiert die üblichen Parameter: einen Zeiger auf das übergeordnete Fenster, die Steuerelement-ID, Position und Größe. Zusätzlich benötigt er einen *BOOL*-Parameter, *Multiline*, und einen *Text*-Parameter vom Typ *LPSTR*. *Multiline* gibt an, ob ein mehrzeiliges Editierfeld erstellt werden soll. Ein weiterer Parameter, *ATextLen* enthält die maximale Anzahl der Zeichen, die in das Editierfeld eingegeben werden können. Da der Text nullterminiert sein muß, ist die Anzahl der Zeichen, die eingegeben werden können, um 1 niedriger als der übergebene *ATextLen*-Parameter.

Im folgenden Programmfragment wird ein einzeiliges und ein mehrzeiliges Editierfeld erstellt. Der Anwender kann in jedes Editierfeld maximal 39 Zeichen eingeben:

```
SingleLineEdit = new TEdit(this, ID_SEDIT, "Default Text",
                           20, 50, 150, 30, 40, FALSE);
MultilineEdit = new TEdit(this, ID_MEDIT, "Default Text",
                          20, 50, 150, 30, 40, TRUE);
```

Die folgenden Stile sind für alle Editierfelder voreingestellt:

WS_CHILD,
WS_VISIBLE,
WS_TABSTOP,
ES_LEFT,
ES_AUTOHSCROLL und
WS_BORDER.

Wenn *Multiline TRUE* ist, so handelt es sich um ein mehrzeiliges Editierfeld, das noch weitere voreingestellte Stile hat:

ES_MULTILINE,
ES_AUTOVSCROLL,
WS_HSCROLL und
WS_VSCROLL.

Nach der Erzeugung des Editierfeldes kann sein Stil geändert werden:

```
BorderlessEdit = new TEdit(this, ID_SEDIT, "Default Text",
                           20, 50, 150, 30, 40, FALSE);
BorderlessEdit->Attr.Style &= ~WS_BORDER;
```

Diese Zeilen produzieren das in Abbildung 12.8 gezeigte Editierfenster, wenn die Elementfunktion *Create* des Editierfeldes aufgerufen wird.

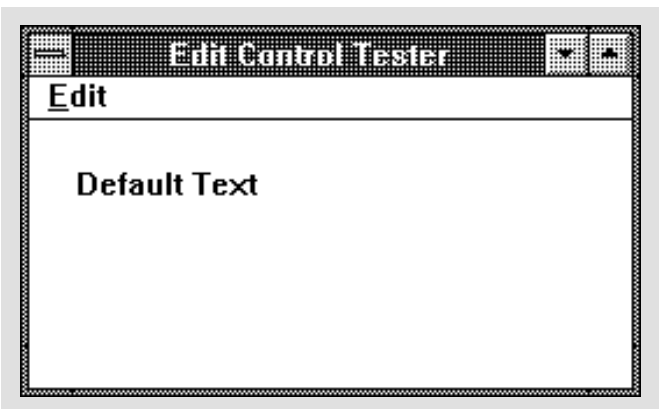


Abbildung 12.8
Ein Editierfeld ohne Rahmen

In der folgenden Tabelle sind gebräuchliche Stile für Editierfelder zusammengestellt.

Tabelle 12.3 Gebräuchliche Stile für Editierfelder

Stil	Ergebnis
ES_LEFT	Ausrichtung links
ES_CENTER	Zentriert
ES_RIGHT	Ausrichtung rechts
ES_MULTILINE	Mehrzeiliges Editierfeld
ES_UPPERCASE	Umwandlung in Großbuchstaben
ES_LOWERCASE	Umwandlung in Kleinbuchstaben
ES_AUTOVSCROLL	Automatischer vertikaler Bildlauf bei mehrzeiligen Editierfeldern
ES_AUTOHSCROLL	Automatischer horizontaler Bildlauf bei mehrzeiligen Editierfeldern
WS_BORDER	Rechteckiger Rahmen um das Editierfeld
WS_VSCROLL	Hinzufügen einer vertikalen Bildlaufleiste
WS_HSCROLL	Hinzufügen einer horizontalen Bildlaufleiste

Zwischenablage und Editieroperationen

Mit den folgenden Elementfunktionen von *TEdit* können Sie Text direkt von einem Editierfeld in die Zwischenablage von Windows übergeben und umgekehrt.

void Copy();
void Cut();
void Paste();

Außerdem stehen folgende Editier-Elementfunktionen zur Verfügung:

Bool CanUndo();
void Clear(); (von TStatic geerbt)
Bool DeleteSelection();
void Undo();

Die Elementfunktion *CanUndo* gibt an, ob die letzte Änderung rückgängig gemacht werden kann.

Sie werden dem Anwender häufig über ein *Edit*-Menü den Zugriff auf diese Elementfunktionen ermöglichen. Ein Editierfeld kann unmittelbar auf die Auswahl von Menüpunkten wie *Edit/Copy* und *Edit/Undo* antworten. *TEdit* definiert auf Befehlen basierende Elementfunktionen, wie zum Beispiel *CMEditCopy* und *CMEditUndo*, die als Antwort auf eine bestimmte Menüauswahl (eine Befehlsbotschaft) im übergeordneten Fenster des Editierfeldes aufgerufen werden. *CMEditCopy* ruft *Copy* und *CMEditUndo* ruft *Undo* auf.

In Tabelle 12.4 sind die Elementfunktionen aufgeführt, die als Antwort auf Menüauswahlen mit einer bestimmten Menü-ID (definiert in rcowl.h) aufgerufen werden: Sie müssen nur ein *Edit*-Menü hinzufügen, dessen Einträge die oben aufgeführten Menü-IDs besitzen. Sie müssen keine Elementfunktion schreiben.

Tabelle 12.4 Menü-IDs und die von ihnen aufgerufenen Elementfunktionen

Menü-ID	Aufgerufene Elementfunktion
CM_EDITCUT	*CMEditCut*
CM_EDITCOPY	*CMEditCopy*
CM_EDITPASTE	*CMEditPaste*
CM_EDITCLEAR	*CMEditClear*
CM_EDITDELETE	*CMEditDelete*
CM_EDITUNDO	*CMEditUndo*

Informationen über Editierfelder

Sie werden ein Editierfeld immer dann abfragen, wenn Sie den Eingabetext auf seine Korrektheit überprüfen müssen oder wenn die Eingabe für eine spätere Verwendung gespeichert oder in ein anderes Steuerelement kopiert werden soll. *TEdit* unterstützt eine Vielzahl von Elementfunktionen, die der Abfrage und

Bearbeitung dienen. Viele Abfrage- und Bearbeitungs-Elementfunktionen liefern eine Zeilennummer oder die Position eines Zeichens in der Zeile zurück bzw. erfordern deren Angabe. Alle Indexpositionen beginnen bei 0. Mit anderen Worten: Die erste Zeile ist die Zeile 0 und das erste Zeichen in einer Zeile ist das Zeichen 0. Die wichtigsten Abfrage-Elementfunktionen sind *GetText*, *GetLine*, *GetNumLines* und *GetLineLength*.

Die Abfrage-Elementfunktionen von *TEdit*, die Text aus einem Steuerelement zurückliefern, erhalten auch die Textformatierung. Dies spielt nur bei mehrzeiligen Editierfeldern eine Rolle, in denen Text in mehr als einer Zeile angezeigt wird. In diesem Fall enthält der Text zwei zusätzliche Zeichen für jeden erforderlichen Zeilenumbruch: ein *Carriage Return* (0x0D) und ein *Linefeed* (0x0A). Wenn der Text wieder aus der Zwischenablage zurück in ein Editierfeld gebracht, in eine Datei geschrieben oder auf einem Drucker ausgegeben wird, erscheinen die Zeilenumbrüche genauso wie im Editierfeld. Deshalb müssen Sie bei der Verwendung einer Abfrage-Elementfunktion, die eine Zeichenanzahl ermittelt, immer die beiden zusätzlichen Zeichen für den Zeilenumbruch berücksichtigen.

GetText liefert den Text aus einem Editierfeld und füllt den String, auf den das übergebene *LPSTR*-Argument zeigt, mit dem Inhalt des Editierfeldes bis zur Zeichenanzahl, die im zweiten Parameter festgelegt ist, einschließlich der Zeilenumbrüche. Der Rückgabewert des Aufrufs ist die Anzahl der Zeichen, die kopiert wurden, ohne abschließendes Nullzeichen. Wenn das Editierfeld keinen Text enthält, wird Null zurückgeliefert. Die folgende Prozedur holt Text aus einem Editierfeld und liefert ihn im *TextBuffer*-Argument zurück:

```
void TTestWindow::ReturnText(LPSTR TextBuffer, int BufferSize)
{
  char TheText[20];

  if (BufferSize >= sizeof(TheText) &&
       (Edit->GetText(TheText, sizeof(TheText) != 0))
    strcpy(TextBuffer, TheText);
  else strcpy(TextBuffer, "")
}
```

GetLine hat eine speziellere Wirkung als *GetText*. Es liefert in einem mehrzeiligen Editierfeld den Text aus der Zeile, die im Integer-Argument angegeben wird. Zeile 0 ist die erste Zeile.

GetNumLines liefert die Anzahl der Zeilen in einem mehrzeiligen Editierfeld. Sie sollten *GetNumLines* verwenden, wenn Sie überprüfen wollen, wieviele Zeilen in ein Editierfeld eingegeben wurden, bevor Sie Text mit *GetLine* abrufen.

GetLineLength liefert die Anzahl der Zeichen in der angegebenen Zeile eines mehrzeiligen Editierfeldes. Wenn die Zeile existiert, aber keine Zeichen enthält, liefert *GetLineLength* 0 zurück. *GetLineLength* sollte zusammen mit *GetNumLines* und *GetLine* verwendet werden, um Text aus einem mehrzeiligen Editierfeld abzurufen:

```
void TTestWindow::ReturnLine(int LineNum, LPSTR TextBuffer,
                             int BufferSize){
  char TheText[20];
```

```
    if ((Edit->GetNumLines >= LineNum) &&
        (Edit->GetLineLength(LineNum) >= sizeof(TheText)) &&
        (Edit->GetLine(TheText, sizeof(TheText) != 0))
      strcpy(TextBuffer, TheText);
    else
      strcpy(TextBuffer, "")
}
```

GetSubText ist ebenfalls spezialisierter als *GetText*. Sie benötigt ein *LPSTR*- und zwei Integer-Argumente, die den Start- und Endwert eines Textbereichs in einem Editierfeld festlegen. Das erste Zeichen hat die Position 0. In einem mehrzeiligen Editierfeld wird der Index sequentiell von der ersten Zeile an hochgezählt. Zeilenumbrüche zählen als zwei Zeichen. Dadurch kann Ihr Programm das Format des Textes im Editierfeld rekonstruieren, wenn der Text angezeigt, gedruckt, in ein anderes Editierfeld übertragen oder in die Zwischenablage gebracht wird.

Sie werden sich nun fragen, wie die Indizes ermittelt werden, die als Argumente im Aufruf von *GetSubText* übergeben werden. Eine Möglichkeit ist die Verwendung von *GetSelection*, einer Elementfunktion, die Start- und Endwert des aktuell ausgewählten (hervorgehobenen) Bereichs zurückliefert. Normalerweise wird dieser Bereich vom Anwender ausgewählt, dies kann aber auch vom Programm mit *SetSelection* durchgeführt werden. (Vergleichen Sie dazu den folgenden Abschnitt »Editierfelder ändern«.) Die letzte Indexposition, die von *GetSelection* zurückgeliefert wird, ist die Position des letzten ausgewählten Zeichens plus 1.

Sie können in einer Anwendung eventuell Zeit sparen, die für die weitere Verarbeitung von Text aus Editierfeldern erforderlich ist. Die *IsModified*-Elementfunktion, eine *BOOL*-Funktion, liefert den Wert *TRUE*, wenn der Anwender den Text in einem Editierfeld geändert hat, und *FALSE*, wenn der Text unverändert ist. Die Rückgabe von *FALSE* kann Ihnen den Aufruf von *GetText* und *GetLine* ersparen. *ClearModify* setzt das Flag wieder auf *FALSE*.

GetLineIndex und *GetLineFromPos* sind zwei Elementfunktionen, mit deren Hilfe Sie die Position von Text in einem mehrzeiligen Editierfeld berechnen können. *GetLineIndex* liefert die Anzahl der Zeichen (einschließ lich der beiden Zeichen für Zeilenumbrüche) in allen Zeilen, die vor der im Integer-Argument angegebenen Zeile liegen. Die Elementfunktion liefert alle Zeichen des Editierfeldes, wenn die angegebene Zeile nicht existiert. *GetLineFromPos* benötigt einen Index für eine Zeichenposition und liefert die Zeilennummer, die zu dieser Position gehört. Beide Elementfunktionen sind dann hilfreich, wenn Sie die Struktur der Zeilen und die Inhalte eines mehrzeiligen Editierfeldes abfragen wollen.

Editierfelder ändern

In einem herkömmlichen Programm zur Dateneingabe besteht für das Programm keine Veranlassung, ein Editierfeld direkt zu ändern. Der Anwender ändert den Text und das Programm liest den Text mit einer Abfrage-Element-

funktion. Da Editierfelder auch für viele andere Zwecke verwendet werden, müssen auch Anwendungen selbst Text ersetzen, einfügen, löschen oder auswählen können.

In einer ObjectWindows-Anwendung können Elementfunktionen der Instanzen der Klasse *TEdit* aufgerufen werden, die eine vom Programm gesteuerte Textmanipulation in Editierfeldern ausführen.

Text löschen

Clear löscht einfach den gesamten Text in einem Editierfeld. Sie können *Clear* als Teil einer Reset-Operation verwenden, die einige oder alle Editierfelder in einem Fenster löscht. *DeleteSelection* löscht den aktuell ausgewählten Text eines Editierfeldes. Ein Texteditor kann den Zugriff auf *Clear* und *DeleteSelection* über Menüauswahlpunkte zur Verfügung stellen (vgl. »Zwischenablage und Editieroperationen« weiter vorne in diesem Kapitel). *DeleteSelection* ist eine *BOOL*-Elementfunktion, die *FALSE* zurückliefert, wenn kein Text ausgewählt ist.

DeleteSubText ähnelt *DeleteSelection*. Sie löscht jedoch den Text, der zwischen der übergebenen Anfangs- und Endposition steht. Der Text wird inklusive des ersten, aber ohne das letzte Zeichen gelöscht.

DeleteLine löscht den gesamten Text, der in der angegebenen Zeile steht – jedoch nicht die zwei Zeichen für den Zeilenumbruch. Alle anderen Zeilen bleiben unverändert.

Text einfügen

Insert hat eine ähnliche Funktion wie *Paste*. *Insert* verwendet als einzufügenden Text nicht den Inhalt der Zwischenablage, sondern das übergebene Argument. Sie löscht vorhandenen ausgewählten Text, hebt jedoch den eingefügten Text nicht hervor. Mit *Insert* können Sie Ihre eigene Zwischenablage (nur für Text) im Programm implementieren.

SetText kombiniert die Aktionen von *Clear* und *Insert*. Sie löscht den gesamten Inhalt des Editierfeldes und fügt den Text des übergebenen Arguments ein. Wenn Sie beispielsweise einen Bestell-Eingabebildschirm für Bestellungen neu setzen möchten, die in erster Linie aus Kalifornien kommen, könnten Sie *SetText* wie folgt aufrufen:

```
StateField->SetText("CA");
```

und damit das *state*-Eingabefeld setzen.

Textauswahl und Bildlauf

SelectSelection bewirkt die Auswahl oder Markierung (optische Hervorhebung) von Text zwischen der übergebenen Anfangs- und Endposition, wobei jedoch das Zeichen an der Endposition nicht eingeschlossen wird.

Scroll führt den Bildlauf in einem mehrzeiligen Editierfeld aus. *Scroll* benötigt zwei Integer-Argumente: die Anzahl der Zeichen, um die horizontal, und die Anzahl der Zeilen, um die vertikal verschoben wird. Eine positive Integerzahl verschiebt nach rechts bzw. nach unten, eine negative Integerzahl nach links bzw. nach oben. Nur mehrzeilige Editierfelder können einen vertikalen Bildlauf durchführen.

Beispielprogramm: EditTest

EditTest ist ein Programm mit einem Hauptfenster, das als übergeordnetes Fenster für zwei Editierfelder, zwei statische Steuerelemente und einen Schalter dient. Abbildung 12.7 stellt dieses Fenster dar.

Wenn der Anwender den Schalter anklickt, wird der Text aus dem linken Editierfeld (*Edit1*) in das rechte (*Edit2*) kopiert. Der Text wird in *Edit2* in Großbuchstaben umgewandelt, weil diesem Feld der Stil *ES_UPPERCASE* zu den voreingestellten Stilen im Datenelement *Attr.Style* hinzugefügt wurde. Ist in *Edit1* kein Textbereich ausgewählt, wird der gesamte Text in *Edit2* kopiert. Ist in *Edit1* Text ausgewählt, wird nur dieser Teil kopiert.

Das *Edit*-Menü unterstützt Editierfunktionen für das Editierfeld, das gerade den Fokus hat.

Das vollständige Programm finden Sie unter dem Namen EDITTEST.CPP im Unterverzeichnis OWL\EXAMPLES.

Aktionsschalter-Steuerelemente

Die Aktionsschalter (manchmal auch »Befehlsschalter« genannt) führen bei jeder Betätigung bestimmte Aufgaben durch. Es gibt zwei Stile für Aktionsschalter, die beide von *TButton* abgeleitet sind: *BS_PUSHBUTTON* und *BS_DEFPUSHBUTTON*. Beide Stile können für Instanzen von *TButton* verwendet werden. Voreingestellte Aktionsschalter sind Aktionsschalter mit einem dicken Rahmen und stehen für voreingestellte Anwenderantworten.

Der *TButton*-Konstruktor hat die üblichen Parameter des Steuerelement-Konstruktors: das übergeordnete Fenster, die Steuerelement-ID sowie Position und Größe. Zusätzlich hat der *TButton*-Konstruktor noch einen *BOOL*-Parameter *IsDefaultButton* und einen *Text*-Parameter vom Typ *LPSTR*. *IsDefaultButton* zeigt an, ob der Schalter ein voreingestellter oder regulärer Aktionsschalter ist. *Text* enthält den Text für den Schalter. Die folgende Anweisung aus den Konstruktoren des übergeordneten Fensters des Schalters erzeugt einen Schalter mit der Aufschrift "Test Button":

```
Push1 = new TButton(this, ID_BUTTON, "Test Button", 38, 48,
                    316, 24, FALSE);
```

Benachrichtigungsbotschaften beantworten

Sobald der Anwender einen Aktionsschalter anklickt, empfängt das übergeordnete Fenster eine Benachrichtigung vom Schalter. Das übergeordnete Fenster beantwortet diese Botschaft, indem es eine auf der ID des untergeordneten Fensters basierende Botschaftsantwortfunktion definiert.

Im folgenden Beispiel antwortet *TTestWindow::HandleButtonMsg* auf die Benachrichtigungen eines Schalters mit mit der ID *ID_BUTTON*, indem es ein Meldungsfenster anzeigt:

```
class TTestWindow : public TWindow {
public:
  PTRadioButton Button;
  virtual void HandleButtonMsg(RTMessage Msg) =
    [ID_FIRST + ID_BUTTON];

void TTestWindow::HandleButtonMsg(RTMessage Msg)
{
  MessageBox(HWindow, "clicked", "The button was:", MB_OK);
}
```

Windows kennt für Aktionsschalter nur einen Benachrichtigungscode: *BN_CLICKED*. Sie müssen ihn daher nicht überprüfen.

Auswahl- und Schaltfelder

Wir fassen Auswahl- und Schaltfelder manchmal unter dem Oberbegriff *Wahlfelder* zusammen, da beide Feldtypen Anwendereingaben in Form einer Auswahl abrufen. Auswahl- und Schaltfelder haben zwei »Zustände«: »ausgewählt« oder »nicht ausgewählt«. Spezielle Auswahlfelder verfügen noch über einen dritten Status, den Status »nicht auswählbar« (»grau«).

Auswahlfelder dienen dem Anwender normalerweise zur Markierung von Optionen (auch mehreren). Schaltfelder werden dagegen für die Auswahl einer von mehreren Optionen verwendet, die sich gegenseitig ausschließen. So können Auswahlfelder beispielsweise für die Auswahl mehrerer Schriften eingesetzt werden, die eine Anwendung laden soll. Schaltfelder dagegen werden für die Bestimmung einer Schrift für ein einzelnes Zeichen verwendet.

Auswahl- und Schaltfelder werden von Instanzen der ObjectWindows-Klassen *TCheckBox* und *TRadioButton* repräsentiert. *TCheckBox* ist von *TButton*, *TRadioButton* von *TCheckBox* abgeleitet.

Auswahl- und Schaltfelder erzeugen

Die Konstruktoren von *TCheckBox* und *TRadioButton* benötigen die üblichen Konstruktor-Parameter: das übergeordnete Fenster, die Steuerelement-ID, Position und Größe. Außerdem haben beide Konstruktoren einen *Text*-Parameter vom Typ *LPSTR* und einen *AGroup*-Parameter vom Typ *PTGroupBox*. *Text* zeigt auf den Text, den Windows per Voreinstellung auf der rechten Seite des Wahlfeldes anzeigt. *AGroup* ist ein Zeiger auf ein *TGroupBox*-Objekt, das Wahlfelder logisch gruppiert. Wenn *AGroup* den Wert *NULL* hat, ist das Wahlfeld

nicht Teil einer logischen Gruppe (vgl. den Abschnitt »Gruppen« weiter hinten in diesem Kapitel).

```
GroupBox = new TGroupBox(this, ID_GROUPBOX, "A Group Box",
                         38, 102, 176, 108);
CheckBox = new TCheckBox(this, ID_CHECKBOX, "Check Box Text",
                         235, 12, 150, 26, GroupBox);
```

Per Voreinstellung setzt der *TCheckBox*- bzw. *TRadioButton*-Konstruktor die »automatischen« Stile *BS_AUTOCHECKBOX* und *BS_AUTORADIOBUTTON*. Aufgrund dieser Stile schaltet Windows sofort auf den Status »ausgewählt«, wenn ein Wahlfeld angeklickt wird.

Wenn ein Schaltfeld mit einem »automatischen« Stil angeklickt wird, schaltet Windows alle anderen Schaltfelder in der Gruppe aus. Wenn Sie *Attr.Style* überschreiben und die »nicht automatischen« Stile *BS_CHECKBOX* und *BS_RADIOBUTTON* festlegen, müssen Sie selbst für den Status »ein« oder »aus« der Wahlfelder sorgen.

Sollen die Text-Strings auf der linken Seite der Auswahl- und Schaltfelder stehen, müssen Sie vor deren Erstellung den Stil *BS_LEFTTEXT* in *Attr.Style* einfügen:

```
CheckBox->Attr.Style |= BS_LEFTTEXT;
```

Informationen über den Status von Wahlfeldern

Die Abfrage eines Wahlfeldes ist eine Möglichkeit, seinen Status herauszufinden und darauf zu antworten. Schalt- und Auswahlfelder haben zwei »Zustände«: »ausgewählt« und »nicht ausgewählt«. Mit der Elementfunktion *GetCheck* der Klasse *TCheckBox* können Sie den Status eines Wahlfeldes abfragen:

```
MyState = CheckBox->GetCheck();
```

Der Rückgabewert von *GetCheck* kann mit den drei definierten Konstanten *BF_UNCHECKED*, *BF_CHECKED* und *BF_GRAYED* zur Feststellung des Feld-Status verglichen werden.

Wahlfeld-Status ändern

Das Ändern des Wahlfeld-Status scheint die Aufgabe des Anwenders, nicht die Ihres Programms zu sein. Aber in einigen Fällen muß das Programm den Status eines Wahlfeldes direkt ändern können.

Dies ist beispielsweise dann der Fall, wenn Sie Optionen anzeigen wollen, die vorher gewählt und gespeichert wurden. Die Klasse *TCheckBox* definiert vier Elementfunktionen für das Ändern des Status eines Auswahlfeldes: *Check*, *Uncheck*, *Toggle* und *SetCheck* (die allgemeinste Elementfunktion).

Check bewirkt, daß der Status des Auswahlfeldes auf »ausgewählt« gesetzt wird:

```
CheckBox->Check();
```

Uncheck schaltet den Status auf »nicht ausgewählt«:

```
CheckBox->Uncheck();
```

Toggle ändert den Status von »ausgewählt« auf »nicht ausgewählt« oder umgekehrt:

```
CheckBox->Toggle();
```

Im Fall von dreistufigen Schaltern ändert *Toggle* ein nicht ausgewähltes Feld auf »ausgewählt«, ein ausgewähltes Feld auf »nicht auswählbar« und ein nicht auswählbares Feld auf »nicht ausgewählt« um.

SetCheck ändert den Status des Auswahlfeldes beliebig:

```
CheckBox->SetCheck(BF_UNCHECKED);  // Ausgewählt
CheckBox->SetCheck(BF_CHECKED);    // Nicht ausgewählt
CheckBox->SetCheck(BF_GRAYED);     // Nicht auswählbar (»grau«)
```

Wenn Sie ein Schaltfeld mit dem Stil *BS_AUTORADIOBUTTON* durch den Aufruf einer dieser Elementfunktionen auswählen, wird der Status aller anderen ausgewählten Schaltfelder in derselben Gruppe auf »nicht ausgewählt« gesetzt. Lesen Sie dazu den Abschnitt »Gruppen« weiter hinten in diesem Kapitel. Per Konvention sollte in einer Gruppe immer ein Schaltfeld ausgewählt sein.

Auswahl- und Schaltfeldbotschaften beantworten

Wählt der Anwender ein Auswahl- oder Schaltfeld aus, wird eine Benachrichtigungsbotschaft mit dem Code *BN_CLICKED* an dessen übergeordnetes Fenster übergeben. Wie bei Instanzen anderer Steuerelementklassen, werden diese Botschaften normalerweise durch Botschaftsantwortfunktionen verwaltet, die auf der ID des untergeordneten Fensters basieren. Sie können aber auch Klassen von *TButton* oder *TCheckBox* ableiten, deren Instanzen nach dem Anklicken bestimmte Aktionen ausführen.

Gruppen

Eine Windows-Gruppe ist ein Steuerelement in Form eines beschrifteten Rechtecks, das andere Steuerelemente optisch gruppiert. Das logische Gruppieren von Steuerelementen (von dem das gegenseitige Ausschließen von automatischen Schaltfeldern abhängt) beruht nicht auf der Position der Elemente in einer Gruppe, sondern hängt von der Reihenfolge, in der die Steuerelemente erstellt werden, und von der Verwendung des Erstellungsstils *WS_GROUP* ab.

Dennoch sind die *TGroupBox*-Objekte keine »Dummies«, denn sie spielen eine wichtige Rolle bei der logischen Gruppierung von Steuerelementen. Wenn Sie ein *TGroupBox*-Objekt für Ihre Wahlfelder bei deren Erzeugung festlegen, können Sie eine »Gruppen«-Botschaftsantwortfunktion für das übergeordnete Fenster definieren. Dadurch müssen Sie nicht für jedes Element eine separate Botschaftsantwortfunktion erstellen. Dies ist deshalb möglich, weil *TGroupBox* Benachrichtigungen von den Wahlfeldern in seiner Gruppe empfangen und diese in »Gruppen«-Benachrichtigungen für das übergeordnete Fenster überset - zen kann.

Gruppen erzeugen

Der *TGroupBox*-Konstruktor benötigt die üblichen Parameter: das übergeord - nete Fenster, die Steuerelement-ID, Position und Größ e. Zusätzlich wird ein *Text*-Parameter vom Typ *LPSTR* übergeben, der die Beschriftung der Gruppe enthält:

```
GroupBox = new TGroupBox(this, ID_GROUPBOX, "A Group Box", 38,
                         102, 176, 108);
```

Gruppenbotschaften beantworten

Wenn sich der Status eines Auswahlfeldes, das zu einer Gruppe gehört, geändert hat, wird *TGroupBox* davon benachrichtigt. *TGroupBox* sendet dann eine »Gruppen«-Benachrichtigungsbotschaft an das übergeordnete Fenster. Das übergeordnete Fenster kann auf diese Botschaft durch die Definition einer auf der ID des untergeordneten Fensters basierenden Elementfunktion antworten, die die Benachrichtigungen der Gruppe bearbeitet.

Im Beispielprogramm GBOXTEST.CPP wird eine Elementfunktion des über - geordneten Fensters definiert, die auf Auswahländerungen in jeder Gruppe von Schaltfeldern antwortet. GBOXTEST.CPP finden Sie in dem Unterverzeichnis OWL\EXAMPLES.

Beispielprogramm: BtnTest

BtnTest ist ein vollständiges Programm, das ein Fenster mit Aktionsschalter-, Auswahlfelder-, Schaltfelder- und Gruppen-Steuerelementen erstellt. Beim Start der Anwendung wird das Hauptfenster mit den Steuerelementen angezeigt. Wenn der Anwender diese anklickt, gibt die Anwendung die entsprechenden unterschiedlichen Antworten (vgl. Abbildung 12.9).

Das vollständige Programm finden Sie unter dem Namen BTNTEST.CPP im Unterverzeichnis OWL\EXAMPLES.

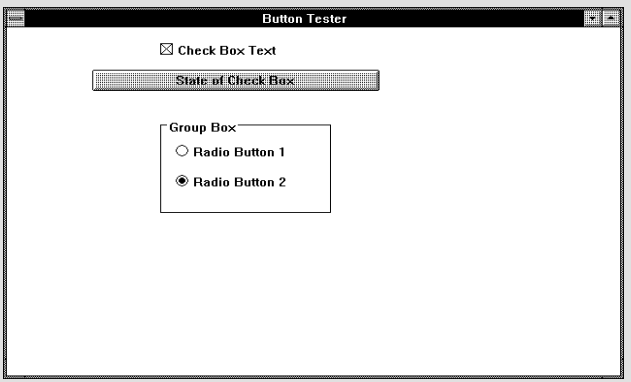


Abbildung 12.9
Ein Fenster mit verschiedenen Schaltern

Bildlaufleisten

Bildlaufleistensteuerelemente verfügen über ein schmales Kästchen, den Positionszeiger, das über die Länge der Bildlaufleiste an verschiedene Positionen verschoben werden kann. Jede Position ist mit einem in einem speziellen Bereich festgelegten Wert verknüpft. Die Bildlaufleistensteuerelemente werden für das Ermitteln eines Wertes verwendet, der in einen vom Anwender gesetzten Bereich liegt. So können diese Elemente z.B. Temperatureinstellungen oder einen Farbwert bestimmen.

Bildlaufleistensteuerelemente sehen zwar aus wie Fenster-Bildlaufleisten, haben aber ansonsten nichts mit ihnen gemeinsam. Diese Bildlaufleisten, die den Bildlauf der Fensteranzeige steuern, sind Bestandteil des Fensters und keine untergeordneten Fenster. Windows erstellt Fenster-Bildlaufleisten für Fenster mit den Erstellungsstilen *WS_HSCROLL* und/oder *WS_VSCROLL*. In ObjectWindows werden Bildlaufleistensteuerelemente auf dieselbe Art und Weise wie andere Steuerelemente erstellt. Sie müssen nur ein Bildlaufleistenobjekt im Konstruktor des übergeordneten Fensters erzeugen.

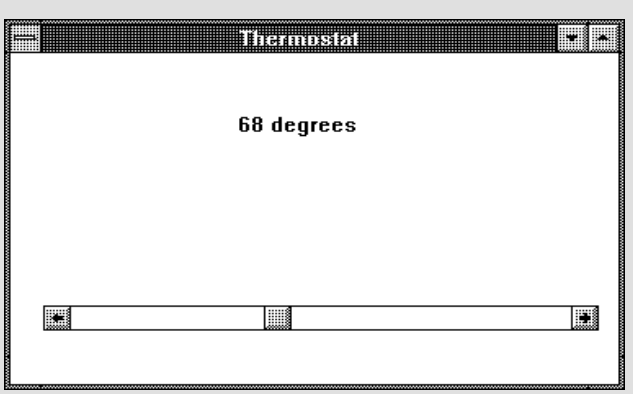


Abbildung 12.10
Ein Bildlaufleistenobjekt

Bildlaufleisten erzeugen

Der *TScrollBar*-Konstruktor benötigt die üblichen Parameter: das übergeordnete Fenster, die Steuerelement-ID, Position und Größe. Außerdem wird ein *BOOL*-Parameter *IsHorizontal* übergeben, der die Ausrichtung (horizontal oder vertikal) der Bildlaufleiste im übergeordneten Fenster angibt. Wenn für die Breite einer vertikalen Bildlaufleiste 0 angegeben ist, wird diese mit einer Standardbreite erzeugt – wie bei einer Liste. Dasselbe gilt, wenn die Höhe einer horizontalen Leiste mit 0 angegeben ist. Die folgende Programmzeile erstellt die horizontale Bildlaufleiste in der Standardhöhe (vgl. Abbildung 12.10):

```
Thermometer = new TScrollBar(this, ID_THERMOMETER, 20, 170,
                             340, 0, TRUE);
```

Der *TScrollBar*-Konstruktor ruft den *TControl*-Konstruktor auf, und fügt dann *SBS_HORZ* oder *SBS_VERT* den für die Bildlaufleiste voreingestellten Stilen hinzu. Sie können noch weitere Stile, wie z.B. *SBS_TOPALIGN* angeben, wenn Sie das Datenelement *Attr.Style* ändern.

In der Abbildung 12.11 sehen Sie verschiedene Bildlaufleistenobjekte.

Der Bereich für die Bildlaufleiste liegt bei ihrer Erstellung per Voreinstellung zwischen 1 und 100. Mit der Elementfunktion *SetRange*, die im Abschnitt »Bildlaufleisten ändern« weiter hinten in diesem Kapitel vorgestellt wird, können Sie den Bereich anders definieren.

Zwei weitere Attribute für Bildlaufleistenobjekte sind die Zeilen- und Seitengrößen. *LineMagnitude* (die Zeilengröße) ist die Anzahl der Positionen, um die der Positionszeiger verschoben wird, wenn der Anwender die Pfeile an den Enden der Bildlaufleiste anklickt. *PageMagnitude* (die Seitengröße) ist die Anzahl der Positionen, um die der Positionszeiger verschoben wird, wenn der Anwender im Bildlaufbereich (zwischen dem Positionszeiger und den Enden der Bildlaufleiste) klickt. *LineMagnitude* und *PageMagnitude* sind per Voreinstellung auf 1 bzw. 10 gesetzt.

Sie können diese Werte neu setzen, indem Sie direkt die Datenelemente *LineMagnitude* und *PageMagnitude* von *TScrollBar* ändern.

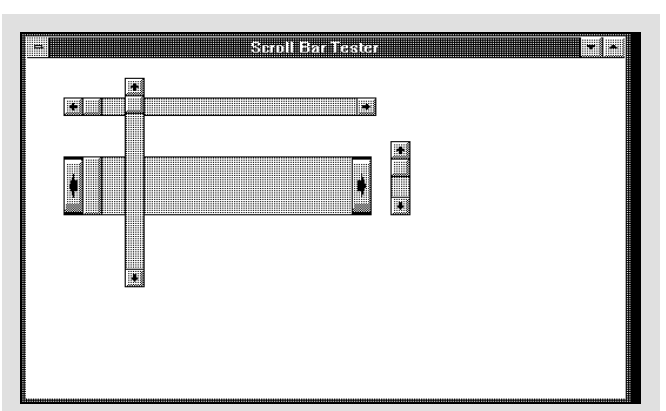


Abbildung 12.11
Ein Fenster mit verschiedenen Bildlaufleisten

Informationen über Bildlaufleisten

TScrollBar definiert zwei Elementfunktionen, die Informationen über Bildlaufleisten liefern: *GetRange* und *GetPosition*. Die *GetRange*-Elementfunktion übernimmt zwei Referenz-Argumente vom Typ Integer. Die Funktion speichert in diesen Integern die höchste und niedrigste Position des Positionszeigers im Bildlaufbereich.

GetPosition ist eine Funktion, die die Position des Positionszeigers als Integerwert zurückliefert. Ihr Programm wird häufig Bereich und Position ermitteln und miteinander vergleichen.

Bildlaufleisten ändern

Das Ändern der Bildlaufleisten liegt in den meisten Fällen in der Hand des Anwenders. Aber auch Ihr Programm kann eine Bildlaufleiste ändern.

SetRange ist eine Elementfunktion, die zwei Integer-Argumente für die höchste und niedrigste Position im Bereich benötigt. Per Voreinstellung liegt der Bereich für neue Bildlaufleisten zwischen 1 und 100. Sie können diesen Bereich aber neu definieren, um eine bessere Anpassung an die tatsächlich verwendeten Schrittweiten der Bildlaufleiste zu erzielen. Dazu rufen Sie nach der Erstellung der Bildlaufleiste deren Elementfunktion *SetRange* auf. Am besten aufgehoben ist dieser Aufruf in der *SetupWindow*-Elementfunktion des übergeordneten Fensters. Achten Sie darauf, daß vor dem Aufruf von *SetRange* die *SetupWindow*-Elementfunktion der Basisklasse des übergeordneten Fensters aufgerufen wird, die das Bildlaufleistenobjekt erstellt.

Im folgenden Beispiel wird der Bereich der Bildlaufleiste, die als Thermometer verwendet wird, in der *SetupWindow*-Elementfunktion des übergeordneten Fensters geändert:

```
void TParentWindowType::SetupWindow()
{
  TWindow::SetupWindow();
  Thermometer->SetRange(32, 120);
}
```

Wenn Ihre Bildlaufleiste eine Instanz der Klasse ist, die Sie von *TScrollBar* abgeleitet haben, ist es selbstverständlich besser, wenn Sie ihren Bereich in ihrer eigenen *SetupWindow*-Elementfunktion ändern.

SetPosition ist eine Elementfunktion mit einem Integer-Argument für die Position, an die Sie den Positionszeiger verschieben wollen. In der im vorhergehenden angesprochenen Thermometer-Anwendung könnte Ihr Programm die Temperatureinstellung beispielsweise direkt auf 78 Grad setzen:

```
Thermometer->SetPosition(78);
```

Die dritte Elementfunktion, *DeltaPos*, verschiebt die Position des Positionszeigers um den im Integer-Argument angegebenen Umfang nach oben (bzw.

nach links) oder unten (bzw. nach rechts). Eine positive Zahl bewegt den Positionszeiger nach unten (rechts). Eine negative Zahl bewirkt ein Verschieben nach oben (links). Wenn Sie zum Beispiel die Temperatureinstellung um 5 Grad verringern wollen, verwenden Sie:

```
Thermometer->DeltaPos(-5);
```

Bildlaufereignisse beantworten

Beim Bildlauf empfängt das übergeordnete Fenster eine entsprechende Benachrichtigungsbotschaft. Wenn Ihr Fenster auf Bildlaufereignisse antworten soll, legen Sie auf die bekannte Weise Botschaftsantwortfunktionen an, die auf der ID des untergeordneten Fensters basieren.

Benachrichtigungen von Bildlaufleisten unterscheiden sich jedoch etwas von den übrigen Benachrichtigungsbotschaften von Steuerelementen. Sie basieren auf den Windows-Botschaften *WM_HSCROLL* und *WM_VSCROLL* und nicht auf *WM_COMMAND*. Der einzige Unterschied, auf den Sie achten müssen, liegt darin, daß die Benachrichtigungscodes von Bildlaufleisten in *Msg.WParam* und nicht in *Msg.LP.HI* gespeichert werden.

Zu den am meisten verwendeten Codes gehören: *SB_LINEUP*, *SB_LINEDOWN*, *SB_PAGEUP*, *SB_PAGEDOWN*, *SB_THUMBPOSITION* und *SB_THUMBTRACK*.

Oft wird die Reaktion auf alle Ereignisse im Überprüfen der neuen Position des Positionszeigers und der entsprechenden Antwort beruhen. Normalerweise werden Sie den Benachrichtigungscode ignorieren. Sehen Sie sich dazu das folgende Beispiel an:

```
void TTestWindow::HandleThermMsg(RTMessage Msg)
{
  int NewPos;

    NewPos = Thermometer->GetPosition();
    // Fortfahren - basieren auf NewPos
}
```

Eventuell wollen Sie erst dann auf das Verschieben des Positionszeigers antworten, wenn der Anwender eine neue Position ausgewählt hat.

Blenden Sie hierzu die *SB_THUMBTRACK*-Botschaften aus:

```
void TTestWindow::HandleThermMsg(RTMessage Msg)
{
  int NewPos;

  if (Msg.WParam != SB_THUMBTRACK)
  {
    NewPos = Thermometer->GetPosition();
    //  Fortfahren - NewPos berücksichtigen
  }
}
```

Eventuell soll das Bildlaufleistenobjekt selbst auf die Benachrichtigung der Bildlaufleiste antworten. Bauen Sie zu diesem Zweck ein spezielles Antwortverhalten in das Bildlaufleistenobjekt ein. Dazu definieren Sie für das Bildlaufleistenobjekt eine der Benachrichtigung entsprechende Botschaftsantwortfunktion für dessen Klasse.

Verwenden Sie die Summe von *NF_FIRST* und den Benachrichtigungscode der Bildlaufleiste als Index für die Botschaftsantwortfunktion. Leiten Sie dazu eine neue Klasse von *TScrollBar* ab:

```
class TSpecializedSBar : public TScrollBar
{
public:
  virtual void SBTop(RTMessage Msg) =
    [NF_FIRST + SB_TOP];
};
TSpecializedSBar::SBTop(RTMessage Msg)
{
  TScrollBar::SBTop(Msg);
  // Sonderbehandlung, wenn der Positionszeiger an den Beginn
  // verschoben wird
}
```

Beachten Sie, daß die Elementfunktion *SBTop* zuerst *TScrollBar::SBTop* aufruft. *TScrollBar::SBTop* verschiebt den Positionszeiger, wenn eine *SB_TOP*-Benachrichtigung empfangen wird.

TScrollBar definiert folgende Elementfunktionen, die auf die entsprechenden Benachrichtigungen antworten:

Tabelle 12.5 Von TScrollBar definierte Elementfunktionen

Elementfunktion	Benachrichtigungsbotschaft
SBLineUp	*SB_LINEUP*
SBLineDown	SB_LINEDOWN
SBPageUp	SB_PAGEUP
SBPageDown	SB_PAGEDOWN
SBThumbPosition	SB_THUMBPOSITION
SBThumbTrack	SB_THUMBTRACK
SBTop	SB_TOP
SBBottom	SB_BOTTOM

Wenn Sie eine dieser Elementfunktionen überschreiben, rufen Sie die *TScrollBar*-Elementfunktion auf, die Sie für das Verschieben des Positionszeigers überschrieben haben.

Beispiel: SBarTest

Das Programm *SBarTest* erstellt die Thermometer-Anwendung, die Sie in der Abbildung 12.10 sehen.

Sie finden das vollständige Programm unter dem Namen SBARTEST.CPP im Unterverzeichnis OWL\EXAMPLES.

Steuerelement-Daten übertragen

Zur Verwaltung von komplexen Dialogfenstern oder Fenstern mit vielen untergeordneten Steuerelementen werden Sie normalerweise eine Klassse ableiten, die den Status ihrer Steuerelemente speichert bzw. ermittelt. Der Status eines Steuerelements beinhaltet den Text eines Editierfeldes, die Position eines Positionszeigers und die Information, ob ein Schaltfeld ausgewählt ist. Sie können aber auch die Definition einer abgeleiteten Klasse umgehen, indem Sie eine Struktur definieren, die den Status der Steuerelemente eines Fensters oder Dialogfensters repräsentiert.

Nehmen wir folgendes Beispiel: Ihr Programm zeigt ein modales Dialogfenster an und kopiert, nachdem es geschlossen wurde, Daten über den Status der Steuerelemente aus dem Transferpuffer. Wenn der Anwender das Dialogfenster wieder aktiviert, werden die Steuerelemente auf den Status gesetzt, der beim letzten Schließen des Dialogfensters gültig war. Zusätzlich können Sie den anfänglichen Status jedes Steuerelements gemäß den Daten im Transferpuffer bestimmen.

Sie können auch jederzeit Informationen in beiden Richtungen explizit übertragen, um beispielsweise den Status von Steuerelementen mit den vorhergehenden Werten wiederherzustellen.

Einen Transferpuffer definieren

Der Transferpuffer ist eine Struktur, die für jedes vom Transfervorgang betroffene Steuerelement ein Feld zur Verfügung stellt. Es gibt aber auch Steuerelemente, die nicht vom Transfermechanismus beeinflußt werden. Dies gilt für Aktionsschalter, die keinen Status haben und für Gruppen.

Zur Definition eines Transferpuffers legen Sie für jedes vom Transfer betroffene Steuerelement im Dialog oder Fenster ein Datenelement an. Es ist aber nicht notwendig, für alle Steuerelemente Datenelemente anzulegen, sondern nur für diejenigen, in die Werte übertragen oder aus denen Werte kopiert werden sollen.

Der folgende Transferpuffer speichert Transferdaten für jeden Steuerelement-Typ.

```
struct SampleTransferStruct {
  char StaticText[STATICTEXTLEN];    // statischer Text
  char EditText[EDITTEXTLEN];        // Text des Editierfeldes
  TListBoxData ListBoxData;          // Listenstrings und Auswahl
  TComboBoxData ComboBoxData;        // Listenstrings des
                                     // Kombinationsfensters
                                     // und Auswahl
  WORD CheckState;                   // Status des Auswahlfeldes
  WORD RadioState;                   // Status des Schaltfeldes
  TScrollBarStruct ScrollBarStruct;  // Bildlaufbereich, etc.
}
```

Wie Sie sehen, bestimmt der Typ des Steuerelements den Typ des Datenelements, das für den Transferpuffer definiert wird. Denn jeder Steuerelementtyp muß unterschiedliche Informationen speichern:

- Statische Steuerelemente und Editierfelder übertragen ihren Text. Ihre *TextLen*-Datenelemente enthalten die Länge des übertragenen Texts zuzüglich einer abschließenden Null.

- Listen übertragen die enthaltenen Strings, den bzw. die ausgewählten Strings und die Anzahl der ausgewählten Strings mit Hilfe eines Objekts des Typs *TListBoxData*. Im Transferpuffer kann ein Zeiger auf ein *TListBoxData*-Objekt angegeben werden:

```
struct MyTransferBuffer
{
  ...
  PTListBoxData pMyListBoxData;
  ...
}
```

pMyListBoxData wird folgendermaßen initialisiert:

```
MyTransferBuffer.pMyListBoxData = new TListBoxData();
```

Die Klassendefinition von *TListBoxData* enthält drei Datenelemente:

Tabelle 12.6 Datenelemente von TListBoxData

Datenelement	Definition
PArray Strings	Ein Array von Strings (aus der Container-Klassenbibliothek) mit den Listeneinträgen.
PArray SelStrings	Ein Array von Strings mit den ausgewählten Listeneinträgen. Bei einer Liste für Einzelauswahl gibt es nur einen String.
Int SelCount	Die Anzahl der ausgewählten Einträge. Bei einer Liste für Einzelauswahl ist *SelCount* 1.

- Kombinationsfenster übertragen die Strings im Listenbereich sowie einen Zeiger auf den gewählten String in einem Objekt des Typs *TComboBoxData*. Sie können im Transferpuffer einen Zeiger auf ein *TComboBoxData*-Objekt angeben:

```
struct MyTransferBuffer
{
  ...
  PTComboBoxData pMyListBoxData;
  ...
}
```

pMyComboBoxData wird folgendermaßen initialisiert:

```
MyTransferBuffer.pMyComboBoxData = new TComboBoxData();
```

Die Klassendefinition von TComboBoxData enthält zwei Datenelemente:

Tabelle 12.7 Datenelemente von TComboBoxData

Datenelement	Definition
PArray Strings	Ein Array von Strings (aus der Container-Klassenbibliothek) mit den Einträgen aus dem Listenbereich des Kombinationsfensters.
PChar Selection	Ein Zeiger auf die Zeichenfolge, die der ausgewählte Eintrag in der Liste des Kombinatinsfensters ist.

Auswahl- und Schaltfelder übertragen ihren Status als Werte vom Typ *WORD* (*BF_UNCHECKED*, *BF_CHECKED* und *BF_GRAYED*). Bildlaufleisten verwenden den Record *TScrollBarData* zum Speichern des Bereichs der Bildlaufleiste und der Position des Positionszeigers.

TScrollBarData ist folgendermaßen definiert:

```
struct TScrollBarData {
  int LowValue;
  int HighValue;
  int Position;
}
```

Steuerelemente erzeugen und den Transfer aktivieren

Fenster oder Dialoge, die den Transfermechanismus verwenden, müssen die davon betroffenen Steuerelementobjekte genau in der Reihenfolge erzeugen, in der die entsprechenden Datenelemente im Transferpuffer definiert sind. Zur Aktivierung des Transfermechanismus für ein Fenster- oder ein Dialogobjekt muß nur das Datenelement *TransferBuffer* mit einem Zeiger auf den definierten Transferpuffer belegt werden.

Der Transfermechanismus erfordert den Einsatz von ObjectWindows-Objekten für die Repräsentation der Steuerelemente, für die Sie Daten übertragen wollen. Für die Verknüpfung von Objekten mit Steuerelementen in Dialogfenstern verwenden Sie den Konstruktor, der lediglich das übergeordnete Fenster und einen Ressourcenbezeichner als Argumente benötigt.

Hinweis: Die Verknüpfung von Steuerelementobjekten mit Steuerelementen wird in Kapitel 11 »Dialogobjekte« erläutert.

```
TParentWindow::TParentWindow(PTWindowsObject AParent,
                             LPSTR ATitle) :
                             TWindow(AParent, ATitle)
{
  TheDialog = new TDialog(this, ID_DIALOG);
  new TStatic(TheDialog, ID_STATIC, 20);
  new TEdit(TheDialog, ID_EDIT);
  new TListBox((TheDialog, ID_LISTBOX);
  new TComboBox(TheDialog, ID_COMBOBOX, 20);
  new TCheckBox(TheDialog, ID_CHECKBOX, 0);
  new TRadioButton(TheDialog, ID_RADIOBUTTON, 0);
  new TScrollBar(TheDialog, ID_SCROLLBAR);
  TheDialog->TransferBuffer = &TransferStruct; }
```

Ein weiterer Unterschied zwischen Fenstern und Dialogen liegt darin, daß der Transfermechanismus bei Steuerelementen eines Fensters standardmäßig zu Beginn ausgeschaltet und bei Steuerelementen eines Dialogs eingeschaltet ist. Eine Ausnahme bilden Instanzen von *TStatic*. Hier ist auch bei Steuerelementen eines Dialogs der Transfermechanismus standardmäßig ausgeschaltet.

Zur Aktivierung des Mechanismus rufen Sie die Elementfunktion *EnableTransfer* auf:

```
...
Edit = new TEdit(this, ID_EDIT, "", 10, 10, 100, 30,
                 40, FALSE);
Edit->EnableTransfer();
...
```

Wenn ein Steuerelement explizit vom Transfer ausgeschlossen werden soll, dann rufen Sie nach dessen Erstellung seine *DisableTransfer*-Elementfunktion auf.

Daten übertragen

Bei der Fenster- oder Dialogerstellung und -ausführung werden die Steuerelemente automatisch mit den im Transferpuffer gespeicherten Daten initialisiert.

Für modale Dialoge werden die Informationen standardmäßig nur dann von den Steuerelementen in den Transferpuffer zurück übertragen, wenn der Dialog eine Befehlsbotschaft mit der Steuerelement-ID *ID_OK* empfängt. Die Aktualisierung des Transferpuffers erfolgt, wenn der Aufruf von *CanClose* in *CloseWindow TRUE* zurückliefert. Wird der Dialog dann erneut ausgeführt, werden die Daten im Puffer wieder an die Steuerelemente übertragen. Auf diese Weise bleiben Dialog und Puffer immer synchron.

Sie können jedoch jederzeit Daten in beiden Richtungen übertragen, bei-

spielsweise Informationen aus einem Steuerelement in ein Fenster oder einen nicht-modalen Dialog. Sie können auch den Status der Steuerelemente mit den Daten aus dem Transferpuffer als Antwort auf das Anklicken eines Reset-Schalters wiederherstellen. Verwenden Sie die Elementfunktion *TransferData* mit der Konstante *TF_SETDATA*, um die Daten aus dem Puffer zu den Steuerelementen zu übertragen, und mit der Konstante *TF_GETDATA* für den Transfer in der anderen Richtung. Folgendermaßen könnten Sie *TransferData* zum Beispiel in der *CloseWindow*-Elementfunktion eines Fensterobjekts aufrufen:

```
void TSampleWindow::CloseWindow()
{
  TransferData(TF_GETDATA);
  TWindow::CloseWindow();
}
```

Den Transfer für Steuerelemente anpassen

Sie können auch die Art und Weise ändern, in der ein bestimmtes Steuerelement seine Informationen überträgt oder ein neu definiertes Steuerelement in den Transfermechanismus einbeziehen. In beiden Fällen müssen Sie nur eine *Transfer*-Elementfunktion für Ihr Steuerelementobjekt schreiben, die – wenn *TF_GETDATA* übergeben wird – Daten aus dem Steuerelement an die Position kopiert, deren Adresse übergeben wurde. Wenn *TF_SETDATA* übergeben wird, kopieren Sie die Daten zur übergebenen Adresse in das Steuerelement. Wenn *TF_SIZEDATA* übergeben wird, liefern Sie nur die Größe des Transferpuffers zurück – ohne weitere Aktion. Die Größe des Transferpuffers steht im Datenelement *TextLen* von *TStatic*. Im folgenden sehen Sie als Beispiel *TStatic::Transfer*:

```
WORD TStatic::Transfer(Pvoid DataPtr, WORD TransferFlag)
{
  if (TransferFlag == TF_GETDATA)
    GetText((LPSTR)DataPtr, TextLen);
  else
    if (TransferFlag == TF_SETDATA)
      SetText((LPSTR)DataPtr);
  return TextLen;
}
```

Die *Transfer*-Elementfunktion sollte immer die Anzahl der übertragenen Bytes zurückliefern.

Transfer-Beispiel

Das Hauptfenster des Programms *TranTest* erstellt ein modales Dialogfenster mit Feldern, in die der Anwender Name und Adresse einträgt. Es wird ein

Transferpuffer eingesetzt, der diese Informationen speichert und in den Steuerelementen des Dialogs anzeigt, wenn der Dialog erneut verwendet wird. Beachten Sie, daß keine neue Dialogklasse definiert werden muß, die die Dialogdaten setzt und abruft.

Das vollständige Programm finden Sie unter dem Namen TRANTEST.CPP im Unterverzeichnis OWL\EXAMPLES. Die Beispielprogramme LBXTTEST.CPP und CBXTTEST.CPP demonstrieren den Transfer von Daten einer Liste bzw. eines Kombinationsfensters. Diese Programme sind modifizierte Versionen der Programme LBOXTEST.CPP und CBOXTEST.CPP, die wir weiter vorne in diesem Kapitel vorgestellt haben.

13. Anwenderdefinierte Steuerelementobjekte

In den vorhergehenden Kapiteln wurden die voreingestellten Charakteristika, das Aussehen und die Verhaltensweisen der ObjectWindows-Steuerelementklassen erläutert. Dabei haben wir Ihnen bereits einige Möglichkeiten zur Änderung dieser Voreinstellungen vorgestellt. In diesem Kapitel finden Sie eine Beschreibung der Vorgehensweisen, mit deren Hilfe Sie die vordefinierten ObjectWindows-Steuerelemente modifizieren können. Außerdem stellen wir Ihnen die Abläufe des Entwurfs eigener Steuerelemente vor.

Ein voreingestelltes Steuerelement modifizieren

Es gibt zwei Möglichkeiten zur Änderung der Charakteristika eines vordefinierten ObjectWindows-Steuerelements:

- Die Erstellungsstile des Elements ändern oder
- Botschaftsantwortfunktionen definieren.

Die Erstellungsstile ändern

Sie können das Verhalten und die Eigenschaften eines voreingestellten Steuerelements in gewissem Umfang durch die Änderung seines Stilerstellungsattributs, das in *Attr.Style* gespeichert ist, modifizieren. Wenn Sie beispielsweise verhindern wollen, daß sich Auswahlpunkte in einer Gruppe von Schaltfeldern gegenseitig ausschließen, entfernen Sie *BS_AUTORADIOBUTTON* aus den Erstellungsstilen und fügen stattdessen *BS_RADIOBUTTON* ein. Beachten Sie, daß Sie nun die Schalter selbst überprüfen müssen! Ein weiteres Beispiel: Sie können die Auswahl mehrerer Einträge in einer Liste ermöglichen, indem Sie den Erstellungsstilen der Liste *LBS_MULTIPLESEL* hinzufügen.

Sie können die Erscheinung eines vordefinierten Steuerelements in einem gewissen Umfang ändern, indem Sie seine Erstellungsstile modifizieren. Wenn Sie z.B. bestimmen möchten, daß der Text eines Auswahlfeldes auf der linken Seite angezeigt werden soll, geben Sie folgendes ein:

```
MyBox = new TCheckBox(TheParent, ID_CHECKBOX, "Text", 100, 100,
                      80, 20, NULL);
MyBox->Attr.Style |= BS_LEFTTEXT;
```

Sie können die meisten Steuerelemente mit einem Rahmen versehen, wenn Sie *WS_BORDER* den Erstellungsstilen hinzufügen.

Einigen voreingestellten Steuerelementen können Sie einen benutzergezeichneten Erstellungsstil hinzufügen, um eine größ ere Kontrolle über die Erscheinung des Steuerelements auszuüben. Diese Steuerelemente werden im nächsten Abschnitt erläutert.

Die Änderungen, die Sie an den Charakteristika der vordefinierten Steuerelemente durch die Modifizierung der Erstellungsstile vornehmen können, sind auf die Stiloptionen beschränkt, die für den jeweiligen Steuerelementtyp zur Verfügung stehen. Im *Windows-Referenzhandbuch* finden Sie vollständige Listen der möglichen Erstellungsstile für alle Steuerelementtypen.

Ein voreingestelltes Steuerelement als zeichenbar definieren

Für bestimmte voreingestellte Steuerelementtypen können Sie einen benutzergezeichneten Stil bestimmen, wenn Sie die Erscheinung des Steuerelements (innerhalb des voreingestellten Rahmens) definieren wollen. Um einen Schalter zeichenbar zu machen, geben Sie für ihn den Erstellungsstil *BS_OWNERDRAW* an:

```
Attr->Style |= BS_OWNERDRAW;
```

Auß erdem müssen Sie die Art und Weise festlegen, in der das Steuerelement gezeichnet werden soll. Zu diesem Zweck definieren Sie in einer abgeleiteten Klasse eine Elementfunktion *ODADrawEntire*. Sie können auch die Elementfunktionen *ODAFocus* und *ODASelect* überschreiben, um die Darstellung des zeichenbaren Steuerelements zu ändern, wenn dieses den Fokus erhält oder wenn es ausgewählt wird.

Den Elementfunktionen *ODADrawEntire*, *ODAFocus* und *ODASelect* wird eine *DRAWITEMSTRUCT*-Struktur übergeben, die Informationen für das Zeichnen eines Steuerelements enthält. Ihr Datenelement *hDC* enthält ein Handle auf den Bildschirmkontext, der verwendet werden soll. Das Handle muß an die Windows-GDI-Funktionen übergeben werden, wenn diese das Steuerelement zeichnen. Das Datenelement *rcItem* enthält den Client-Bereich des Steuerelements. In Kapitel 3 »Windows-Typen« im *Windows-Referenzhandbuch* finden Sie eine detaillierte Erläuterung von *DRAWITEMSTRUCT*.

Ein zeichenbares Steuerelement kann auch vom übergeordneten Fenster gezeichnet werden, das *WMDrawitem* überschreibt. Dabei wird in *LParam* der angegebenen *RTMessage* ein Zeiger auf *DRAWITEMSTRUCT* übergeben. Der Bezeichner des Steuerelements wird im *CtlID*-Element der Struktur übergeben.

Die Anwendung DCTLTEST.CPP im Unterverzeichnis OWL\EXAMPLES zeigt einen Dialog mit einem zeichenbaren Schalter an, wenn der Menüeintrag "Test" ausgewählt wird. Führen Sie den Dialog aus, und sehen Sie sich an, wie die zeichenbaren Schalter auf den Fokus und Auswahländerungen antworten. Verschieben Sie mit den Pfeil- oder der Tab -Taste den Fokus und klicken Sie die

Schalter an, um sie auszuwählen. In Ihren eigenen Anwendungen können Sie Bitmaps für Ihre zeichenbaren Schalter verwenden.

Aktionsschalter, Auswahlfelder, Schaltfelder und Gruppen können mit dem Erstellungsstil *BS_OWNERDRAW* als zeichenbar definiert werden. Für Listen und Kombinationsfenster stehen zu diesem Zweck *LBS_OWNERDRAW* und *CBS_OWNERDRAW* zur Verfügung. Für Bildlaufleisten, Editierfelder und statische Steuerelemente stellt Windows diese Möglichkeit nicht zur Verfügung.

Standard-Botschaftsantworten ändern

Sie können eine eigene Botschaftsantwort für jede Windows-Botschaft definieren, die an ein vordefiniertes Steuerelement gesendet wird. Dazu definieren Sie Botschaftsantwortfunktionen in einer Klasse, die von einer vordefinierten Steuerelementklasse abgeleitet ist. Mit Hilfe dieser Technik können Sie das Aussehen, das Verhalten und die Eigenschaften eines vordefinierten Steuerelements ändern.

Zusätzliche Bearbeitungsschritte

Sie können für die an ein vordefiniertes Steuerelement übergebene Botschaft zusätzliche Bearbeitungsschritte ausführen lassen – vor oder nach dem Aufruf von *DefWndProc*, durch den eine Standardantwort aktiviert wird.

Wenn beispielsweise ein Schalter zusätzlich zu der Benachrichtigung seines übergeordneten Fensters einen Signalton abgeben soll, sobald er gedrückt wird, können Sie *WMLButtonDown* überschreiben:

```
void TBeepButton::WMLButtonDown(RTMessage Msg)
{
  MessageBeep(0);
  DefWndProc(Msg);
}
```

Da die Standardbearbeitung eines Schalters für die Botschaft *WM_LBUTTONDOWN* im Senden einer *BN_CLICKED*-Benachrichtigungsbotschaft besteht, kann eine alternative Implementierung durch Überschreiben von *BNClicked* vorgenommen werden.

So könnten Sie z.B. die Elementfunktionen *WMSetFocus* und *WMKillFocus* definieren, damit der Rahmen eines Editierfeldes geändert wird, sobald dieses den Eingabe-Fokus erhält. In diesem Fall besteht die Standardbearbeitung in der Übergabe der Benachrichtigungsbotschaften *EN_SETFOCUS* bzw. *EN_KILLFOCUS*. Hier könnte mit Hilfe der Definition von *ENSetFocus*- und *ENKillFocus*-Elementfunktionen eine alternative Implementierung erreicht werden.

Standard-Botschaftsantwortfunktionen überschreiben

Innerhalb gewisser Grenzen können Sie die Standardbearbeitung für vordefinierte Steuerelemente ganz ersetzen. Dazu müssen Sie mehr tun, als Sie vielleicht denken, denn die Standardbearbeitung für Steuerelemente kann sehr komplex sein. Es ist deshalb unbedingt erforderlich, die Ereignisse einer Ände-

rung zu überprüfen. Unter Umständen müssen Sie auch ein vordefiniertes Verhalten simulieren.

Wenn Sie beispielsweise ein Editierfeld implementieren wollen, aus dem nur gelesen werden kann, können Sie eine »aktionsfreie« *WMChar*-Antwort in einer von *TEdit* abgeleiteten Klasse definieren. Danach zeigt das Editierfeld Text an, der in die Zwischenablage kopiert werden, aber nicht geändert werden kann.

Ein weiteres Beispiel für das Überschreiben eines Standardverhaltens: Sie können die Standardbearbeitung von *WMLButtonDown* für einen Schalter ändern, damit ein »flacher« Schalter implementiert wird, der sein Aussehen beim Anklicken nicht ändert. Sie müssen dazu an das übergeordnete Fenster eine Benachrichtigungsbotschaft mit einem *BN_CLICKED*-Code senden, wie dies normalerweise in der vordefinierten Bearbeitung durch *WMLButtonDown* geschieht:

```
void TFlatButton::WMLButtonDown(RTMessage Msg)
{
  SendMessage(Parent->HWindow, WM_COMMAND, Attr.Id,
              MAKELONG(HWindow, BN_CLICKED));
}
```

Statt sehr viele Antworten zu ändern oder anzupassen, ist es sinnvoller, eigene Steuerelemente zu implementieren. Mehr dazu im folgenden Abschnitt.

Verwendung eines anwenderdefinierten Steuerelements

Wenn das Verhalten und die Eigenschaften eines von Ihnen benötigten Steuerelements nicht weitgehend denjenigen eines vordefinierten Steuerelements entspricht, sollten Sie ein eigenes Element verwenden. Dazu müssen dessen Aussehen und sein gesamtes Verhalten definiert werden.

Ein Steuerelement entwerfen

Folgende Arbeitsschritte müssen Sie für den Entwurf eines anwenderdefinierten Steuerelements ausführen:

- Zu ermittelnde Anwendereingabe festlegen.
- Die sichtbare Erscheinung des Steuerelements definieren.
- Die Antworten auf die Eingaben bestimmen.

Ein einfaches Steuerelement antwortet auf das Drücken der linken Maustaste, indem es eine Benachrichtigungsbotschaft an das übergeordnete Fenster sendet. Sie können dieses Verhalten für Ihre Anwendung vollständig ändern. Verlieren Sie aber nicht aus den Augen, daß das Ziel eine leicht zu bedienende und konsistente Schnittstelle für den Anwender sein muß. Dazu sollten Sie einige Konventionen beachten.

Ein Steuerelement definieren

Folgende Arbeitsschritte müssen Sie für die Implementierung eines anwenderdefinierten Steuerelements ausführen:

- Eine von *TControl* abgeleitete Klasse definieren.
- *Paint* für das Zeichnen des Steuerelements überschreiben.
- Botschaftsantwortfunktionen, die die Anwendereingaben abfangen und beantworten, definieren.

Im Unterverzeichnis OWL\EXAMPLES finden Sie eine Beispielanwendung für ein anwenderdefiniertes Steuerelement mit dem Namen CCTLTEST.CPP. Das Steuerelement stellt eine Farboption zur Verfügung, die vom Anwender ausgewählt werden kann. Ein *TColorControl*-Objekt sendet als Antwort auf das Klicken der linken Maustaste eine Benachrichtigungsbotschaft an sein übergeordnetes Fenster:

```
void TColorControl::WMLButtonDown(RTMessage)
{
  SendMessage(Parent->HWindow, WM_COMMAND, Attr.Id,
              MAKELONG(HWindow, CN_CLICKED));
}

void TColorControl::WMLBUTTONDblClk(RTMessage)
{
  SendMessage(Parent->HWindow, WM_COMMAND, Attr.Id,
              MAKELONG(HWINDOW, CN_DBLCLICKED));
}
```

Die Konstanten *CN_CLICKED* und *CN_DBLCLICKED*, die an das übergeordnete Fenster des Farbensteuerelements übergeben werden, sind Benachrichtigungscodes, die den Typ des eingetretenen Ereignisses (Einzel- oder Doppel-Klick) beschreiben. Anwenderdefinierte Steuerelemente senden Benachrichtigungsbotschaften mit anwenderdefiniertem Code. Ein anwenderdefiniertes Steuerelement kann jeden Code senden, den sein übergeordnetes Fenster versteht.

Da *CN_CLICKED* und *CN_DBLCLICKED* keine Standard-Benachrichtigungscodes sind, müssen Sie sie selbst definieren:

```
const unsigned int CN_CLICKED = 0;
const unsigned int CN_DBLCLICKED = 1;
```

TColorControl in der CCTLTEST-Anwendung überschreibt *Paint*, so daß der Client-Bereich mit einem Pinsel der Farbe gezeichnet wird, die von *TColorControl* repräsentiert wird. Sie sollten *Paint* auch für Ihr anwenderdefiniertes Steuerelement überschreiben – ein nicht sichtbares Steuerelement ist nämlich nicht sehr hilfreich! Sie können auch die Botschaftsantwortfunktionen *WMSetFocus* und *WMKillFocus* überschreiben, damit sich die Darstellungsform des Steuerelements ändert, wenn es den Fokus erhält oder abgibt.

TControl erbt die Elementfunktion *GetWindowClass* von *TWindow*, die die Registrierungsattribute bestimmt, die als Voreinstellung für ein anwenderdefiniertes Steuerelement geeignet sind. Wenn Sie diese Voreinstellungen ändern

wollen, müssen Sie *GetWindowClass* und *GetClassName* überschreiben. *TColorControl* ändert beispielsweise den Stil der Registrierungsklasse so, daß das Steuerelement Doppel-Klick-Botschaften empfangen muß:

```
void TColorControl::GetWindowClass(WNDCLASS& AWndClass)
{
  TControl::GetWindowClass(AWndClass);
  AWndClass.style |= CS_DBLCLKS;
}

LPSTR TColorControl::GetClassName()
{
  return "ColorControl";
}
```

Im Abschnitt »Windows-Fensterklassen-Registrierung« in Kapitel 10 »Fensterobjekte« finden Sie Informationen über diese voreingestellten Attribute und das Überschreiben von *GetWindowClass* und *GetClassName*.

Sie müssen den Fensterklassennamen eines anwenderdefinierten Dialogsteuerelements in der Definition der Dialogressource festlegen. Dabei handelt es sich um denselben Namen, der von der Elementfunktion *GetClassName* zurückgeliefert wird. Wenn Sie *GetClassName* nicht überschrieben haben, wird "OWLWindow" zurückgeliefert.

14. MDI-Objekte

Die Multidokumentschnittstelle (MDI) ist ein Schnittstellenstandard für Windows-Anwendungen, in denen der Anwender gleichzeitig mit mehreren geöffneten Dokumenten arbeiten kann. Unter einem Dokument verstehen wir in diesem Zusammenhang eine von Dateien abhängige Aufgabe wie das Editieren einer Textdatei oder die Arbeit in einer Kalkulationsdatei. In MDI-Anwendungen kann der Anwender viele Dateien gleichzeitig öffnen. Beispiele für bekannte MDI-Anwendungen sind der Programm-Manager und der Datei-Manager von Windows. Der MDI-Standard ist Teil der IBM-Spezifikation *Common User Access* (CUA), im Deutschen unter der Abkürzung SAA (System-Anwendungs-Architektur) bekannt.

Die Komponenten einer MDI-Anwendung

Bestimmte Komponenten sind in jeder MDI-Anwendung vorhanden. Die wichtigste ist das Hauptfenster, das in diesem Zusammenhang Rahmenfenster heiß t. Im Client-Bereich des Rahmenfensters steht ein nicht sichtbares Fenster, das MDI-Client-Fenster, das die Fenster verwaltet, die als untergeordnete MDI-Fenster bezeichnet werden.

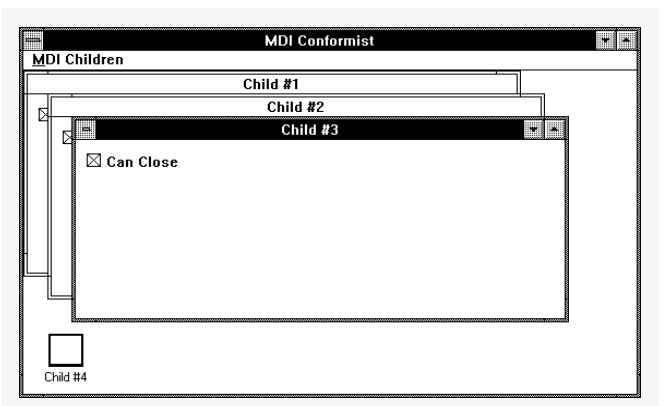


Abbildung 14.1 Die Komponenten einer MDI-Beispielanwendung

Das Rahmenfenster verfügt über ein Menü, das häufig den Titel *Window* trägt und das Einträge wie *Tile*, *Cascade*, *Arrange Icons* und *Close All* enthält. Für deutsche Programmversionen heißen der Menü-Titel *Fenster* und die Einträge *Nebeneinander*, *Überlappend*, *Symbole anordnen* und *Alle schließen*. Jedes geöffnete untergeordnete MDI-Fenster wird am Ende dieses Menüs angefügt. Das aktuell ausgewählte Fenster ist in diesem Menü mit einem Häkchen gekennzeichnet.

Jedes untergeordnete MDI-Fenster hat einige Eigenschaften von überlappenden Fenstern. Es kann auf die volle Größe des MDI-Client-Fensters vergrößert oder zu einem Symbol innerhalb des Rahmenfensters verkleinert werden. Untergeordnete MDI-Fenster erscheinen niemals außerhalb ihres Rahmenfensters. Ein untergeordnetes MDI-Fenster kann kein Menü haben, so daß alle Funktionen über das Menü des Rahmenfensters gesteuert werden. Als Titel eines untergeordneten MDI-Fensters dient häufig der Name der zugehörigen geöffneten Datei.

Diese Eigenschaft ist jedoch optional und steht unter Programmkontrolle. Sie können sich eine MDI-Anwendung als »Mini«-Windows-Sitzung mit vielen durch Fenster oder Symbole repräsentierten Anwendungen vorstellen.

Das MDI-Fenster – ein Objekt

ObjectWindows definiert Klassen, deren Instanzen das MDI-Rahmenfenster und die Client-Fenster darstellen. Dabei handelt es sich um *TMDIFRAME* und *TMDIClient*. Die untergeordneten MDI-Fenster sind Instanzen einer Klasse, die von *TWindow* abgeleitet ist.

In einer ObjectWindows-MDI-Anwendung ist das MDI-Client-Fenster seinem Rahmenfenster zugehörig. Im Datenelement *ClientWnd* des Rahmenfensters ist eine Referenz auf das MDI-Client-Fenster gespeichert. Das Rahmenfenster enthält alle seine untergeordneten Fenster in seiner Liste *ChildList*.

Die *TMDIFrame*-Elementfunktionen befassen sich in erster Linie mit der Erzeugung und Verwaltung der untergeordneten MDI-Fenster und des MDI-Client-Fensters sowie mit der Bearbeitung der Menüauswahlpunkte. Die wichtigste Aufgabe von *TMDIClient* ist die selbständige Verwaltung der untergeordneten MDI-Fenster. Beim Entwurf von MDI-Anwendungen werden normalerweise neue Klassen für Rahmen- und untergeordnete Fenster von *TMDIFrame* und *TWindow* abgeleitet. Meist werden Sie für Ihr MDI-Client-Fenster eine Instanz von *TMDIClient* verwenden.

MDI-Fenster erzeugen

Bei der Erzeugung von Fenstern für eine MDI-Anwendung müssen Sie einige Besonderheiten beachten.

Eine vollständige Erläuterung der Erstellung von Fenstern finden Sie in Kapitel 10 »Fensterobjekte«.

MDI-Rahmenfenster erzeugen

Das Rahmenfenster einer MDI-Anwendung ist gleichzeitig das Hauptfenster der Anwendung. Es wird aus diesem Grund in der *InitMainWindow*-Elementfunktion seines Anwendungsobjekts erzeugt. Anders als bei *TWindow* hat der Konstruktor von *TMDIFrame* kein übergeordnetes Fenster als Argument: Rahmenfenster sind Hauptfenster und haben kein übergeordnetes Fenster. Dafür hat er einen weiteren Parameter, nämlich eine Ressourcen-ID für das Menü. MDI-Rahmenfenster erfordern ein Menü, weshalb der *TMDIFrame*-Konstruktor ein Menüargument benötigt und *Attr.Menu* für Sie setzt.

Das Menü des Rahmenfensters muß über ein dem MDI-Stil entsprechendes Menü für untergeordnete Fenster verfügen. Hier handelt es sich um das Menü, an dessen Ende die Menüeinträge der untergeordneten MDI-Fenster angefügt werden. Das Datenelement *ChildMenuPos* von *TMDIFrame* enthält die Position des untergeordneten Fensterelements. Der Konstruktor von *TMDIFrame* setzt *ChildMenuPos* zu Beginn auf 0, womit der am weitesten links stehende Hauptmenüeintrag bezeichnet wird. Sie können *ChildMenuPos* jedoch im Konstruktor von *TMDIFrame* neu setzen:

```
TMyMDIWindow::TMyMDIWindow(LPSTR ATitle, HMENU AMenu) :
                           TMDIFrame(ATitle, AMenu)
{
  ChildMenuPos = 1;
}
```

TMDIFrame::SetupWindow ruft *InitClientWindow* auf, um das *TMDIFrame*-Objekt zu initialisieren, das als MDI-Client-Fenster fungiert. *TMDIFrame::SetupWindow* erstellt das MDI-Client-Fenster.

Untergeordnete MDI-Fenster erzeugen

TMDIFrame definiert eine Botschaftsantwortfunktion mit dem Namen *CreateChild*. Sie wird aufgerufen, wenn der Anwender einen Menüpunkt mit einer *CM_CREATECHILD*-ID auswählt. Dieser Menüeintrag heißt normalerweise *New* (bzw. *Neu*) oder *Create*. Da *CreateChild* von *TMDIFrame* definiert wird, erzeugt sie ein untergeordnetes MDI-Fenster durch den Aufruf von *TMDIFrame::CreateChild*. *CreateChild* ruft dann zur Erstellung des Fensters *InitChild* und die Elementfunktion *MakeWindow* des Anwendungsobjekts auf.

TMDIFrame::InitChild erstellt ein Fenster vom Typ *TWindow*. Da dieses Fenster keinen Titel hat, können Sie *InitChild* für die MDI-Fensterklasse überschreiben, um eine Instanz der untergeordneten MDI-Fensterklasse zu erzeugen:

```
PTWindowsObject TMyMDIWindow::InitChild()
{
  return new TMyMDIChild(this, "New Child Window");
}
```

Nun wird durch einen Aufruf von *CreateChild* eine Instanz der untergeordneten MDI-Fensterklasse erzeugt und erstellt. Eventuell möchten Sie, daß das Rahmenfenster zu Beginn *ein* untergeordnetes MDI-Fenster anzeigen soll. Untergeordnete MDI-Fenster müssen in der *SetupWindow*-Elementfunktion des MDI-Rahmenfensters erzeugt und erstellt werden und nicht, wie andere untergeordnete Fenster, in dessen Konstruktor. Sie müssen also das untergeordnete Fenster explizit erstellen:

```
void TMyMDIWindow::SetupWindow()
{
  TMDIFrame::SetupWindow();
  CreateChild();
}
```

Soll dieses erste untergeordnete MDI-Fenster maximiert werden, erzeugen Sie das untergeordnete MDI-Fensterobjekt entweder direkt oder durch den Aufruf von *InitChild*. Fügen Sie dann *WS_MAXIMIZED* den bereits für das Objekt gesetzten Stilen hinzu. Danach rufen Sie die Elementfunktion *MakeWindow* des Anwendungsobjekts zur Erstellung des untergeordneten MDI-Fensters auf:

```
void TMyMDIWindow::SetupWindow()
{
  PMyMDIChild NewChild;

  TMDIFrame::SetupWindow();
  NewChild = new TMyMDIChild(this, "New Child Window");
  NewChild->Attr.Style |= WS_MAXIMIZE;
  GetApplication()->MakeWindow(NewChild);
}
```

In manchen Anwendungen sollen untergeordnete MDI-Fenster als Reaktion auf die Auswahl nicht nur eines Menüpunkts erstellt werden. Beispielsweise zeigt ein Texteditor ein neues untergeordnetes Fenster mit dem Dateinamen als Titel an, wenn der Anwender *New* oder *Open* wählt. Definieren Sie für diesen Fall eine Befehlsantwortfunktion für jeden Menüeintrag, der ein untergeordnetes Fenster erzeugen und erstellen soll. Zum Verständnis der Begriffe »Erzeugen« (bzw. »Initialisieren«) und »Erstellen eines Fensters« sehen Sie in Kapitel 10 »Fensterobjekte« nach.

Botschaften in MDI-Anwendungen bearbeiten

Wie bei regulären über- und untergeordneten Fenstern gelangen Befehlsbotschaften zuerst zum untergeordneten Fenster, damit sie dort abgefangen und bearbeitet werden können. Danach werden diese Botschaften an das übergeordnete Fenster weitergegeben. Auf diese Weise kann mit Hilfe des Rahmenfenstermenüs die Aktivität im aktuellen untergeordneten MDI-Fenster gesteuert werden. Das Rahmenfenster hat ebenfalls die Möglichkeit, auf Auswahlen aus seinem Menü zu antworten.

Untergeordnete MDI-Fenster bearbeiten

Die MDI-Fensterklassen von ObjectWindows verfügen über Elementfunktionen zur Manipulation der untergeordneten MDI-Fenster in einer MDI-Anwendung. Obwohl ein sehr großer Teil der auszuführenden Aufgaben von *TMDIClient* ausgeführt wird, kann über die *TMDIFrame*-Elementfunktionen auf alle Funktionen und Daten zugegriffen werden.

Aktivierung von untergeordneten Fenstern

Der Anwender einer MDI-Anwendung kann jedes geöffnete oder zum Symbol verkleinerte untergeordnete MDI-Fenster aktivieren. Sie wollen aber vielleicht eine Aktion ausführen, wenn der Anwender ein Fenster durch Aktivierung eines anderen deaktiviert. So könnten die Menüs des Rahmenfensters den aktuellen Status des aktiven untergeordneten Fensters durch graue Einfärbung oder eine Markierung anzeigen. Jedesmal, wenn ein untergeordnetes Fenster aktiviert oder deaktiviert wird, empfängt es die Windows-Botschaft *WM_MDIACTIVATE*. Sie können die Elementfunktion *TMDIClient::WMMDIActivate* (sie ruft normalerweise *DefWndProc* auf) für das untergeordnete Fensterobjekt überschreiben, damit es entsprechend antwortet.

Das Menü für untergeordnete Fenster

TMDIFrame definiert Botschaftsantwortfunktionen, die auf die Standardauswahlpunkte *Tile*, *Cascade*, *Arrange Icons* und *Close All* des MDI-Menüs antworten. Diese Elementfunktionen erwarten Befehlsbotschaften mit vordefinierten Menü-ID-Konstanten. Stellen Sie sicher, daß diese IDs beim Aufbau einer Menüressource für das untergeordnete Fenster verwendet werden:

Tabelle 14.1 Standard-MDI-Aktionen, Befehle und Elementfunktionen

Aktion	Menü-ID-Konstante	*TMDIFrame*-Elementfunktion
Tile (Nebeneinander)	*CM_TILECHILDREN*	*CMTileChildren*
Cascade (Überlappend)	*CM_CASCADECHILDREN*	*CMCascadeChildren*
Arrange Icons (Symbole anordnen)	*CM_ARRANGEICONS*	*CMArrangeIcons*
Close All (Alle schließen)	*CM_CLOSECHILDREN*	*CMCloseChildren*

Die Botschaftsantwortfunktionen von *TMDIFrame*, wie zum Beispiel *CMTileChildren*, rufen andere *TMDIFrame*-Elementfunktionen auf, wie zum Beispiel *TileChildren*. Diese Elementfunktionen rufen *TMDIClient*-Elementfunktionen desselben Namens auf, wie zum Beispiel *TMDIClient::TileChildren*. Um Verhal-

ten und Eigenschaften neu zu definieren (was aber normalerweise nicht nötig ist), überschreiben Sie *TMDIFrame::TileChildren* bzw. andere *TMDIFrame*-Elementfunktionen (aber nicht die automatischen Botschaftsantwortfunktionen).

Beispiel für eine MDI-Anwendung

Das Programm *MDITest* erzeugt die MDI-Anwendung, die Sie in Abbildung 14.1 sehen.

Das vollständige Programm finden Sie unter dem Namen MDITEST.CPP im Unterverzeichnis OWL\EXAMPLES.

15. Streams und Objekte

Dieses Kapitel befaß t sich damit, wie die Stream-Verwaltung von ObjectWindows für den *Fortbestand* von Objekten sorgt. Die verschiedenen Objekte, die Sie während einer ObjectWindows-Anwendung erstellen können (wie Fenster, Dialoge, Listen, Kollektionen, etc.), haben nur eine bestimmte Lebensdauer. Sie werden im Verlauf einer Anwendung initialisiert, angezeigt, verwendet und wieder freigegeben. Objekte kommen und gehen, so wie das Programm Gültigkeitsbereiche durchläuft. Bei Beendigung des Programms verschwinden sie völlig. Mit Hilfe der Stream-Verwaltung können Sie diese Objekte im Speicher oder in Datei-Streams speichern, so daß sie über ihre normale Lebensdauer hinaus bestehen bleiben.

Es gibt zahllose Anwendungen die fortbestehende Objekte verlangen. Wenn diese in einem gemeinsam benutzten Speicher stehen, können sie beispielsweise eine Kommunikation zwischen verschiedenen Prozessen herstellen und über Modems auf andere Systeme übertragen. Aber der wichtigste Punkt in diesem Zusammenhang ist, daß sie mit Hilfe von Datei-Streams bleibend auf Diskette/Platte gespeichert werden können. So ist es möglich, daß die Objekte von derselben Anwendung, von anderen Instanzen derselben Anwendung oder sogar von anderen Anwendungen wieder gelesen und in ihrer früheren Form wieder hergestellt werden. Diese effiziente und sichere Stream-Fähigkeit steht allen ObjectWindows-Objekten zur Verfügung. Alle ObjectWindows-Klassen, einschließ lich *TButton*, *TDialog* und *TWindow* sind als stream-fähige Klassen entworfen, so daß ihre Behandlung bei der Ein- und Ausgabe in Streams so einfach ist wie die normale Datei-Ein-/Ausgabe.

Der Aufbau eigener stream-fähiger Klassen ist kein komplizierter Vorgang. Eine stream-fähige Klasse benötigt drei virtuelle Funktionen (*read*, *write* und *streamableName*), eine *build*-Funktion und einen Build-Konstruktor. Die Funktionen *read*, *write* und *streamableName* sind als virtuelle Funktionen in *TStreamable* deklariert. Alle stream-fähigen Klassen müssen direkt oder indirekt von *TStreamable* abgeleitet sein. Da *TStreamable* zu den Basisklassen von *TWindowsObject* gehört, ist jede von *TWindowsObject* abgeleitete Klasse stream-fähig.

Stream-Objekte werden mit *pstream* und davon abgeleiteten Klassen erstellt. Diese Klassen, die vor allem für fortbestehende Stream-Ein-/Ausgaben zur Verfügung stehen, ähneln den *iostream*-Standardklassen von C++. Wenn Ihnen diese Klassen vertraut sind, werden Sie mit dem vorliegenden Thema keinerlei Schwierigkeiten haben. Wir werden Ihnen im folgenden Abschnitt eine kurze Einführung zu diesen neuen Klassen bieten und einige Begriffe abklären.

Die Bibliothek iostream

Weder C noch C++ verfügen über Schlüsselwörter oder vordefinierte Operatoren für die Ein-/Ausgabe. Beide Sprachen stützen sich auf Funktionen in Standardbibliotheken: C auf die Bibliothek *stdio* und C++ auf die Bibliothek *iostream*. Die *iostream*-Bibliothek nützt die Leistungsfähigkeit der objektorientierten Programmierung aus und ist aus diesem Grund flexibler, sicherer und erweiterbar. Sie stellt überschriebene Operatoren und eine typensichere Ein-/Ausgabe sowohl für Standard- als auch für anwenderdefinierte Datentypen zur Verfügung. Obwohl die *stdio*-Funktionen, wie beispielsweise *printf*, in C++ verfügbar sind, bevorzugen die meisten C++-Programmierer Streams. Was aber ist ein Stream genau?

Ein Stream ist ein abstrakter Datentyp, der eine (theoretisch) unbegrenzte Abfolge von Einträgen mit unterschiedlichen Zugriffseigenschaften darstellt. Streams verfügen über eine *Länge* (die Anzahl der Elemente), eine *aktuelle Position* (der eindeutige Zugriffspunkt zu jedem beliebigen Zeitpunkt) und einen *Zugriffsmodus* (Nur-Lesen bei Eingabe, Nur-Schreiben bei Ausgabe oder Lesen/Schreiben für Eingabe/Ausgabe). Einlesen (oder *Holen*) findet an der aktuellen Position statt, die dann normalerweise zum nächsten Eintrag weiterrückt, Schreiben (oder *Einfügen*) wird durch Anfügen von Einträgen an das Ende des Streams durchgeführt.

Die traditionelle (Disketten-/Platten-) Datei ist eine bekannte Stream-Implementierung eines Streams. Das zugrundeliegende Konzept wurde jedoch erweitert: es gibt

- Streams im Speicher,
- Zeichen-Streams von der Tastatur (wie zum Beispiel die Standardeingabe *cin*),
- Zeichen-Streams an den Bildschirm (wie die Standardausgaben *cout* und *cerr*),
- Streams zu und von seriellen Anschlüssen und anderen Peripheriegeräten.

Weite Teile dieses Konzeptes stammen aus der UNIX-Welt, wo »alles eine Datei« ist. So kann – aufgrund der objektorientierten Programmierung – eine *Quelle* (oder ein *Produzent*) von Daten mit *Holen*-Elementfunktionen versehen und als Eingabe-Stream behandelt werden. Auf ähnliche Weise ist das *Ziel* (oder der *Konsument*) von Daten mit *Einfügen*-Elementfunktionen ausgestattet und als Ausgabe-Stream »behandelbar«. Streams, die mit Disketten-/Platten-Dateien verknüpft sind, unterstützen normalerweise Ein- und Ausgabe. Im *Borland-C++ Benutzerhandbuch* finden Sie eine Einführung in die C++-Standardbibliothek *iostream*. Diese implementiert eine umfangreiche Hierarchie von Stream-Klassen, die die vielen verfügbaren Varianten verwaltet. Stream-Klassen können eine Kombination folgender Möglichkeiten repräsentieren: gepufferte und nichtgepufferte, formatierte und unformatierte, im Speicher oder in einer Datei befindliche Versionen, wobei jede als Eingabe-, Ausgabe- oder eine Kombination von Ein- und Ausgabe fungieren kann. Aus diesem Grund ist es möglich, daß ObjectWindows für C++ Stream-Ein-/Ausgaben für komplexe Objekte zur Verfügung stellen kann.

Die überschriebenen Operatoren << und >>

Einer der Schlüssel des Erfolgs der C++-Stream-Ein-/Ausgaben ist das Vorhandensein von überschriebenen, verknüpfbaren Operatoren: der Operator << für die Ausgabe (Einfügen) und der Operator >> für die Eingabe (Holen). Die komplexe Syntax der *printf*- und anderer *stdio*-Bibliotheksfunktionen wird durch einfache und elegante Anweisungen der folgenden Art ersetzt:

```
cout << "Hello World" << endl;
```

Da die Operatoren << und >> für einen bestimmten Datentyp und eine Stream-Klasse überschrieben werden können, ist es möglich, Objekte von diesem Typ in die entsprechenden Stream-Objekte zu schreiben oder von dort zu lesen. Für die Standardtypen **char**, **short**, **int**, **long**, **char***, **float**, **double** und **void*** sind in C++ diese Operatoren schon überschrieben. Dies kann vom Programmierer aber auch auf anwenderdefinierte, nicht zu Klassen gehörige Datentypen ausgeweitet werden. Das Überschreiben von Klassenobjekten ist zwar etwas schwieriger, wird aber für Sie von ObjectWindows ausgeführt. Sie können daher Objekte mit Anweisungen wie den folgenden lesen und schreiben:

```
os << AWindow << ADialog;
is >> AButton;
```

ohne sich dabei um interne Details kümmern zu müssen. Es gibt nur einige wenige Aspekte, die Sie bei der Erstellung und *Registrierung* von eigenen streamfähigen Klassen beachten müssen, und die wir Ihnen kurz zusammengefaßt vorstellen wollen. Im obigen Programmfragment ist *os* ein *opstream* (oder abgeleitetes) Objekt und *is* ein *ipstream* (oder abgeleitetes) Objekt. Diese beiden Klassen, die von *pstream* abgeleitet sind, dienen als Basisklassen für fortbestehende Objekte (das *p* in *pstream* steht für persistent, d.h. fortbestehend). Sie befinden sich wie die Klassen *istream* und *ostream* (die von *ios* abgeleitet sind) in der C++-Standardklassen-Hierarchie. Es gibt außerdem noch die Klasse *iopstream*, die eine Kombination aus *ipstream* und *opstream* darstellt. Weiterhin finden Sie Datei-Stream-Varianten namens *ifpstream*, *ofprestream* und *iofpstream*. Diese entsprechen wiederum den C++-Klassen *ifstream*, *ofstream* und *fstream* und verfügen über ähnliche Elemente und Funktionen. Abbildung 15.1 zeigt die Klassenhierarchie von *pstream*.

Überschriebene <<Operatoren, die z.B. für das Schreiben von *TWindow*-Objekten und Objektzeigern in ein *opstream* verwendet werden, sind folgendermaßen in window.h definiert:

```
inline Ropstream operator << (Ropstream os, RTWindow cl);
   { return os << (RTStreamable) cl; }
inline Ropstream operator << (Ropstream os, PTWindow cl);
   { return os << (PTStreamable) cl; }
```

In der ersten Variante wird das *TWindow*-Objekt *cl* in das *opstream*-Objekt *os* geschrieben. Von dem Operator wird derselbe Ausgabe-Stream zurückgeliefert, der die vertraute Verknüpfung von <<-Operationen in der C++-Standardklasse

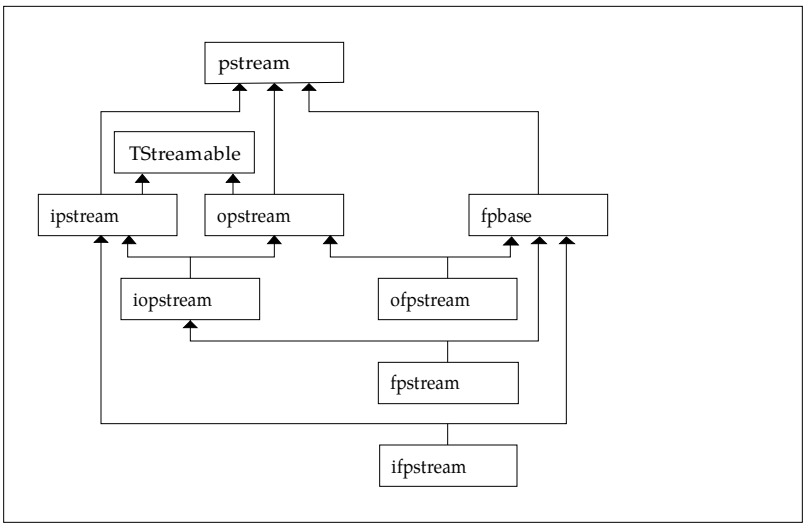


Abbildung 15.1 Hierarchie der streamfähigen Klassen von ObjectWindows

ostream ermöglicht. Die zweite Variante leistet dasselbe für einen Zeiger auf ein *TWindow*-Objekt. Beachten Sie, daß die Operatoren einfach durch Typenumwandlung und den Aufruf des in der Basisklasse definierten <<-Operators implementiert werden. In diesem Fall hat *TWindow* als Basisklasse *TStreamable*.

Überschriebene Operatoren << und >> stehen auch für alle anderen streamfähigen Klassen in ObjectWindows zur Verfügung. Sie sind normalerweise auf die oben gezeigte Art und Weise *inline* definiert. Zum Überschreiben der Operatoren << und >> für Ihre eigenen stream-fähigen Klassen werden Sie normalerweise ähnlichen Programmtext schreiben.

Stream-fähige Klassen und TStreamable

Eine stream-fähige Klasse ist eine Klasse, deren Objekte in fortbestehende Streams geschrieben und von dort gelesen werden können, wobei die Hilfsmittel der ObjectWindow-OWS_-Stream-Verwaltung verwendet werden. Damit geeignete Ein-/Ausgabeoperatoren zur Verfügung stehen (normalerweise handelt es sich dabei um die überschriebenen Operatoren << und >>, Sie können aber auch andere verwenden), muß eine stream-fähige Klasse an beliebiger Stelle in der Hierarchie *TStreamable* als Basisklasse haben. Ein Blick auf die Klassenhierarchie in ObjectWindows zeigt, daß *TWindowsObject* mehrfach abgeleitet ist, und zwar von *Object* und *TStreamable*. Aus diesem Grund erben alle Klassen, die von *TWindowsObject* abgeleitet werden, auch die Vorzüge von *TStreamable*. Alle von *TWindowsObject* abgeleiteten Klassen haben auch überschriebene Operatoren << und >> und sind deshalb ebenfalls stream-fähig. *TScroller* ist ebenfalls (unter anderem) von *TStreamable* abgeleitet und deshalb auch stream-fähig.

Die Stream-Verwaltung

Wenn komplexe Objekte in Streams geschrieben oder von dort gelesen werden, ist ein beträchtlicher Arbeitsaufwand erforderlich, den die Stream-Verwaltung für Sie erledigt. Wenn Datenobjekte in einen Stream geschrieben werden, können sie ohne weitere Komplikationen binär gespeichert und wieder geholt werden. Ein Klassenobjekt aber kann eine beliebige Anzahl verschiedener Datentypen besitzen, einschließlich Zeiger auf andere komplexe Objekte und eventuell sogar Zeiger auf die *vtable* (die Tabelle virtueller Funktionen).

Um diesen und anderen Unwägbarkeiten gerecht zu werden, verwaltet und pflegt die Stream-Verwaltung eine Datenbank mit stream-fähigen Klassen unter Verwendung der Klassen *TStreamableTypes*, *TStreamable* und *TStreamableClass*. Der Overhead beträgt 12 Bytes pro stream-fähiger Klasse. Diese Klassen stellen bestimmten Funktionen die grundlegenden Registrierungs- *read*-, *write*- und *build*-Funktionen zur Verfügung. Damit die Stream-Verwaltung weiß, daß es sich bei einer Klasse um eine stream-fähige Klasse mit einem korrekten Eintrag in der Datenbank handelt, muß die Klasse von ihr registriert werden. Im nächsten Abschnitt finden Sie eine Erläuterung des Build-Prozesses.

Während der Zugriffe auf einen Stream muß die Stream-Verwaltung zusätzlich eine Datenbank mit allen Objekten verwalten, die in einen Stream geschrieben oder von dort gelesen werden. Da Sie sich darum nicht weiter kümmern müssen, wollen wir hier nicht zu sehr ins Detail gehen, uns aber zumindest ansehen, was im Prinzip vor sich geht, wenn Objekte unter Heranziehung von Zeigern in einen Stream geschrieben werden.

Nehmen wir an, *ptr1* und *ptr2* zeigen beide auf dasselbe Objekt und Sie schreiben **ptr1* und **ptr2* in einen Stream. Wenn diese Objekte nacheinander aus dem Stream gelesen werden, erhalten Sie zwei identische Kopien des Objekts mit unterschiedlichen Zeigern – und damit die Basis für ein potentielles »Chaos«. Mit der Datenbank umgeht die Stream-Verwaltung dieses Problem: Es wird nur eine Kopie von **ptr1* in den Stream geschrieben. Ähnlich ist es beim Lesen des Objekts. Wenn ein Objekt aus einem Stream in **ptr1* und **ptr2* gelesen wird, wird nur ein Objekt erstellt, und *ptr1* und *ptr2* zeigen beide auf dieses Objekt.

Die Stream-Verwaltung arbeitet eng mit den Stream-Objekten zusammen, indem sie die entsprechenden *read*- und *write*-Funktionen zur Verfügung stellt. *TStreamable* besitzt abstrakte *read*- und *write*-Funktionen, die in jeder von ihr abgeleiteten Klasse überschrieben werden müssen:

```
class TStreamable {
...
protected:
  virtual Pvoid read( Ripstream ) = 0;
  virtual void write( Ropstream ) = 0;
};
```

Die *write*-Funktion schreibt alle notwendigen Felder des Objekts in den Stream. Jede stream-fähige Klasse muß eine *write*-Funktion haben, wobei die Implementierung aber vereinfacht werden kann, wenn die *write*-Funktion der

Basisklasse verwendet wird. Im folgenden sehen Sie dazu die Definition von *TWindow::write*:

```
void TWindow::write(Ropstream os)
{
  long SaveStyle;
  BOOL NameIsNumeric;
  TWindowsObject::write(os);
  if ( !IsFlagSet(WB_FROMRESOURCE) )
  {
    SaveStyle = Attr.Style & ~(WS_MINIMIZE | WS_MAXIMIZE);
    if ( HWindow )
      if ( IsIconic(HWindow) )
        SaveStyle |= WS_MINIMIZE;
      else
        if ( IsZoomed(HWindow) )
          SaveStyle |= WS_MAXIMIZE;
    os << SaveStyle << Attr.ExStyle << Attr.X
       << Attr.Y << Attr.W << Attr.H
       << (long) (Attr.Param);
  }
  os << Attr.id;

  NameIsNumeric = HIWORD(Attr.Menu) == NULL;
  os << NameIsNumeric;
  if ( NameIsNumeric )
    os << (long) (Attr.Menu);
  else
    os.fwriteString(Attr.Menu);

  os << Scroller;
}
```

Die *read*-Funktion ist ähnlich wie die *write*-Funktion. Jede stream-fähige Klasse muß die abstrakte *read*-Funktion in *TStreamable* überschreiben. Dies erfolgt – wie im obigen Beispiel mit *write* gezeigt – häufig dadurch, daß die *read*-Funktion der Basisklasse auf weitere Felder ausgedehnt wird.

Beim Lesen aus Streams tritt unter bestimmten Umständen ein Sonderfall ein: Wenn Sie ein Objekt aus einem Stream in ein bestehendes Objekt mit geeignetem Typ einlesen, überträgt *read* einfach die entsprechenden Daten und *vtable*-Zeiger. Ist kein Objekt vorhanden, in das gelesen werden kann, muß ein Objekt erzeugt werden, und zwar eine Art Gerüst-Objekt, das vom Stream initialisiert werden kann.

Diese Erzeugung ist die Aufgabe der *build*-Funktion. Die *build*-Funktion ruft einen speziellen *StreamableInit*- oder Build-Konstruktor auf, der genügend Speicherplatz für das Objekt reserviert, die Build-Konstruktoren der Vorfahren aufruft und *vtable* initialisiert. Jede stream-fähige Klasse, die von derartigen Situationen beim Lesen betroffen sein kann, muß einen Build-Konstruktor haben.

read-, *write*- und *build*-Funktionen und Build-Konstruktoren sind für die stream-fähigen Standardklassen von ObjectWindows vordefiniert. Wenn Sie eigene stream-fähige Klassen erstellen wollen, müssen Sie diese Funktionen zur Verfügung stellen.

Konstruktoren stream-fähiger Klassen

Wie bereits erläutert, benötigt eine stream-fähige Klasse, deren Objekte aus Streams gelesen werden sollen, einen speziellen Konstruktor, den BuildKonstruktor. Dieser Konstruktor erfordert ein einziges Argument, *streamableInit*, eine Konstante vom Typ **enum**, die in objstrm.h definiert ist.

Alle stream-fähigen Standardklassen verfügen über diesen Konstruktor, der **public** deklariert sein muß. Er hat folgendes Aussehen:

```
class TMyStreamable : public TBase, public TStreamable {
...
public:
   TMyStreamable(StreamableInit s);
...
};
```

Der Datentyp *StreamableInit* ist ein **enum**-Typ und hat nur ein einziges Element, nämlich *streamableInit*. Wenn Sie ein Objekt mit dem folgenden Konstruktor erstellen:

```
TMyStreamable str(streamableInit);
```

werden die Konstruktoren für enthaltene Zeiger nicht aufgerufen, wie dies bei normalen Konstruktoren der Fall ist. Dieses Vorgehen vereinfacht die Speicherverwaltung.

Wenn Sie ein Objekt aus einem Stream in das Objekt *str* lesen, muß die *read*-Funktion für das Objekt keine unerwünschten Daten aus *str* freigeben:

```
TMyStreamable str(streamableInit):
// Erstellen eines "leeren" str
...
ifpstream ifps("str.sav");  // Datei-Stream mit fortbestehenden
                            // Objekten für Eingabe öffnen
ifps >> str;                // Objekt in str lesen
...
```

Aber Vorsicht: Objekte, die mit dem *streamableInit*-Konstruktor erzeugt wurden, wie beispielsweise *str*, können erst dann »sicher« verwendet werden, wenn sie von einer Stream-Lesen-Operation »initialisiert« wurden.

Die **enum**- und Elementnamen haben keine spezielle Bedeutung: Sie dienen nur als eindeutiges Datentyp-Argument, damit dieser Konstruktor von anderen unterschieden werden kann.

Die Elementfunktion *build* und der Build-Konstruktor sind für eine stream-fähige Klasse folgendermaßen definiert:

```
PTStreamable TMyStreamableClass::build()
{
  return new TMyStreamableClass(streamableInit);
}

TMyStreamableClass::TMyStreamableClass(StreamableInit s) :
  TBaseClass(streamableInit)
{
}
```

Mit anderen Worten: der Build-Konstruktor ruft nur den Konstruktor seiner Basisklasse(n) auf, usw. bis zur letzten Basisklasse.

Namen für stream-fähige Klassen

Jede stream-fähige Klasse muß die *private* virtuelle Funktion *streamableName*, die von *TStreamable* abgeleitet ist, überschreiben. Diese Funktion muß einen eindeutigen Namen für die Klasse als nullterminierten String zurückliefern:

```
class TMyStreambale : public TBase, public TStreamable {
...
private:
   virtual const Pchar streamableName() const
   { return "TMyStreamable"; }
};
```

Für die stream-fähigen ObjectWindows-Klassen ist dies bereits ausgeführt. Wenn Sie aber Ihre eigenen stream-fähigen Klassen definieren, müssen Sie selbst für das Überschreiben sorgen. Dazu veranlassen Sie, daß *streamableName* den Klassennamen – wie oben gezeigt – zurückliefert. Die Stream-Verwaltung verwendet den eindeutigen Namen in den verschiedenen Klassen-Datenbanken.

Verwendung der Stream-Verwaltung

Zur Verwendung der Stream-Verwaltung müssen Sie drei Arbeitsschritte ausführen:

- Die Stream-Verwaltung linken.
- Ein geeignetes Stream-Objekt erstellen.
- Das Stream-Objekt verwenden.

Im folgenden erläutern wird diese Schritte im Detail.

Den Code der Stream-Verwaltung linken

Jede stream-fähige Klasse muß drei Funktionen definieren: *read*, *write* und *build*, die zur Bearbeitung der Klassenobjekte dienen. Die Stream-Verwaltung muß diese Funktionen kennen.

Die Funktionen müssen mit jeder Anwendung, die die Stream-Verwaltung einsetzt, verbunden werden. Diese Verbindung, die die *Registrierung* des Klassennamens durch die Stream-Verwaltung darstellt, wird mit dem Makro *__link* ausgeführt. Dieses Makro sorgt für eine Referenz auf ein Objekt vom Typ *TStreamableClass*. Der Name des Objekts ist per Konvention folgendermaßen aufgebaut sein: *Reg<Klassen_Name>*, wobei *<Klassen_Name>* der Name der stream-fähigen Klasse ist, jedoch ohne das *T* am Beginn.

Angenommen, Sie haben die Klasse *TBase* in BASE.CCP wie folgt definiert:

```
class TBase : public TStreamable
{
  int x;
  int y;
};
```

Außerdem haben Sie die Klasse *TDerived* in MYAPP.CPP definiert, die ein Datenelement als Zeiger auf ein *TBase*-Objekt enthält. Sie müssen dann mit dem Makro *__link* die Stream-Verwaltung anweisen, eine Stream-Unterstützung für *TBase* dazuzubinden.

Die Registrierung der stream-fähigen Klasse und die Definition für *TDerived* sieht folgendermaßen aus:

```
__link(RegBase)
class TFoo : public TStreamable
{

  int z;
  PTBase zz;
};
```

PTBase ist wie üblich ein mit dem *_CLASSDEF*-Makro definierter Zeigertyp auf *TBase*. Verwenden Sie das *__link*-Makro immer dann, wenn ein Datenelement als Zeiger auf eine Klasse eingesetzt wird, die außerhalb der Quelldatei definiert ist.

Ein Stream-Objekt erstellen

Die Erstellung eines *ipstream*- oder *opstream*-Objekts erfordert nur eine Deklaration mit entsprechenden Argumenten, genau wie bei den *iostream*-Klassen. Wenn Sie beispielsweise einen Dialog in einen Datei-Stream mit dem Namen DLG.SAV speichern wollen, müssen Sie ein *ofpstream*-Objekt folgendermaßen deklarieren:

```
TChDirDialog cdlg;
...
// Dialog erstellen

ofpstream of( "dlg.sav" );
// Ausgabedatei-Stream mit fortbestehenden Objekten öffnen

of << cdlg;
// Das Dialogobjekt in den Datei-Stream schreiben
```

Der Konstruktor:

```
ofpstream(PChar filename, int mode = ios::out,
          int prot = filebuf::openprot)
```

wurde mit den angegebenen Voreinstellungen aufgerufen. Die verschiedenen Stream-Konstruktoren und ihre Parameter sind in Kapitel 23 »Stream-fähige Klassen in ObjectWindows« erläutert.

Das Stream-Objekt verwenden

Im folgenden sehen Sie eine typische Lese-Operation:

```
TChDirDialog cdlg (streamableInit);
// Aufruf des Build-Konstruktors; Erstellen des Gerüst-Objekts

ifpstream ifps("dlg.sav");
// Eingabedatei-Stream öffnen

ifps >> cdlg;
// Das Dialogobjekt aus dem Datei-Stream in cdlg lesen
```

Kollektionen und Streams

C++-Container-Klassen sind per Voreinstellung nicht stream-fähig. Der folgende Abschnitt enthält ein Beispiel, wie eine neue Container-Klasse von einer existierenden Container-Klasse abgeleitet und dabei als stream-fähig definiert wird. Sie finden eine schrittweise Erläuterung für die Erstellung einer stream-fähigen Klasse und erfahren, wie jede für die Implementierung erforderliche Komponente von der Stream-Verwaltung verwendet wird. Das Beispiel finden Sie unter dem Namen STEP9.CPP im Verzeichnis OWL\EXAMPLES\STEPS.

Array als stream-fähig definieren

Das Programm verwendet die Klasse *Array* (aus der Container-Klassenbibliothek) zum Speichern einer Kollektion von *TPoint*s (einem Objekt, das in dem

Beispiel definiert ist). Die Klasse *Array* ist selbst nicht stream-fähig, so daß wir eine neue Klasse *TPointArray* definieren werden, die sowohl von *Array* als auch von *TStreamable* mehrfach abgeleitet ist. Alle stream-fähigen Klassen müssen von *TStreamable* abgeleitet werden.

Die stream-fähige Klasse TPointArray

Alle stream-fähigen Klassen müssen eine statische Elementfunktion *build*, einen speziellen Konstruktor und die drei virtuellen Funktionen *read*, *write* und *streamableName* definieren. Die folgende Klassendefinition enthält die Deklarationen für die drei Elementfunktionen:

```
class TPointArray : public Array, public TStreamable {
public:
  TPointArray(int upper, int lower = 0, sizeType aDelta = 0)
  Array(upper, lower, aDelta) {}:
...
public:
  static PTStreamable build();
protected:
  TPointArray(StreamableInit) : Array(50, 0, 50) {};
  virtual void write(Ropstream);
  virtual Pvoid read(Ripstream) ;

private:
  const Pchar streamableName() const {
    return "TPointArray"; }
};
```

Das folgende Programmfragment schreibt ein *TPointArray* in einen Stream:

```
ofpstream os("save.dat");
PTPointArray APointArray;
...
os << APointArray;
```

Obwohl es in vielen Fällen nicht unbedingt erforderlich ist, sollte jede stream-fähige Klasse Hol- und Einfüge-Operationen definieren, die eine Typenumwandlung zwischen dem Objekt und der *TStreamable*-Komponente des Objekts durchführen. Sehen Sie sich dazu das folgende Beispiel an:

```
inline Ripstream operator >> (Ripstream is, RTPointArray cl)
  { return is >> (RTStreamable)cl; }

inline Ripstream operator >> (Ripstream is, RPTPointArray cl)
  { return is >> (RPvoid)cl; )

inline Ropstream operator << (Ropstream os, RTPointArray cl)
  { return os << (RTStreamable)cl; }

inline Ropstream operator << (Ropstream os, RTPointArray cl)
  { return os << (PTStreamable)cl; }
```

Im obigen Programmfragment wird ein Zeiger auf ein *TPointArray* in den Stream *os* geschrieben und damit die vierte *inline*-Definition des Einfügen-Operators (*operator<<*) aufgerufen, was letztlich den Einfügen-Operator mit dem *PTStreamable*-Argument aufruft. Dieser Einfügen-Operator wird durch die Stream-Implementierung definiert und hat einen Aufruf der virtuellen Elementfunktion *write* des Objekts, das als Parameter angegeben ist, zur Folge. In unserem Beispiel wird *TPointArray::write* aufgerufen.

```
void TPointArray::write(Ropstream os) {
  RContainerIterator PointIterator = initIterator();
  os << getItemsInContainer();

  while (int(PointIterator) != 0) {
    RObject PointObject = PointIterator++;
    if (PointObject != NOOBJECT)
    {
      os << ((PTPoint) (&PointObject))->X;
      os << ((PTPoint) (&PointObject))->Y;
    }
  }
  delete &PointIterator;
}
```

TPointArray::write weist dem *Array*-Container einen Iterator zu, schreibt die Anzahl der Elemente im Array in den Stream, iteriert dann über das Array und schreibt dabei jedes Element des Arrays heraus. Am Ende wird das Iterator-Objekt freigegeben. Ein modulareres Vorgehen wäre es, wenn Sie die Klasse *TPoint* als stream-fähig definieren. Danach könnte die Schleife einfach das *TPoint*-Objekt in den Stream schreiben. Dadurch fällt das Herausschreiben der *TPoint*-Datenelemente (*X* und *Y*) weg.

Die Elementfunktion streamableName

Beim Schreiben eines Objekts in einen Stream ruft die Stream-Verwaltung die virtuelle Elementfunktion *streamableName* des Objekts auf, um den Namen festzulegen. Dieser Name wird vor dem Herausschreiben des Objekts in den Stream geschrieben. Danach folgt die Definition der *inline*-Elementfunktion *streamableName*.

```
class TPointArray : public Array, public TStreamable {
...
private:
   const Pchar streamableName() const
     { return "TPointArray"; }
...
};
```

Wenn ein Objekt aus einem Stream gelesen wird, liest die Stream-Verwaltung zuerst die String-Repräsentation des Klassennamens ein. Dieser Name wurde von der Stream-Verwaltung durch den Aufruf des statischen *TStreamableClass*-Konstruktors registriert. Alle stream-fähigen Klassen müssen zur Durchführung der Registrierung eine statische Instanz vom Typ *TStreamableClass* erstellen:

```
TStreamableClass RegPointArray("TPointArray", TPointArray::build,
                               __DELTA(TPointArray));
```

Dieser Konstruktor bestimmt den Namen der Klasse, "TPointArray", die Adresse der *build*-Funktion *TPointArray::build* und den Offset von der Basis des Objekts zur *TStreamable*-Komponente des Objekts. Das Makro *__DELTA* berechnet diesen Offset.

Die Stream-Verwaltung pflegt mit Hilfe dieser statischen Objekte eine Datenbank mit allen stream-fähigen Klassen. Wenn ein Objekt des Typs *TPointArray* aus einem Stream gelesen wird, wird dieses in der Datenbank gesucht, um die Adresse der statischen *build*-Funktion *TPointArray::build* festzustellen.

```
PTStreamable TPointArray::build()
{
  return new TPointArray(streamableInit);
}
```

Alle *build*-Funktionen rufen einfach einen speziellen Konstruktor auf, dessen Argument vom Typ *StreamableInit* ist. Dieser Konstruktor ruft seinerseits den Konstruktor seiner Basisklasse auf, initialisiert aber keine Datenelemente. Auf diese Weise wird ein Objekt mit der korrekten Größe erstellt und seine virtuelle Tabelle ausgefüllt.

Die Definition des Konstruktors finden Sie in der obigen Klassendefinition. Der Typ *StreamableInit* ist vom Typ **enum** und hat nur einen Wert: *streamableInit*. Dieser **enum**-Parameter wählt den speziellen Konstruktor für das Lesen in und das Schreiben aus Streams unter allen möglichen Konstruktoren einer Klasse aus.

Die stream-fähige Funktion read

Der Stream-Verwaltung steht jetzt ein gültiges Objekt zur Verfügung. Er ruft nun die virtuelle Elementfunktion *read* auf, die folgendermaßen definiert ist:

```
Pvoid TPointArray::read(Ripstream is) {
  sizeType NumPoints;
  PTPoint APoint;

  is >> NumPoints;

  for (int i = 0; i < NumPoints; ++i)
  {
    APoint = new TPoint(0, 0);
    is >> APoint->X;
```

```
      is >> APoint->Y;
      add(* (APoint));
    }

    return this;
  }
```

In unserem Beispiel liest die *read*-Funktion von *TPointArray* die Anzahl der Elemente im Array und dann die X-, Y-Wertpaare ein, die mit jedem *TPoint* verknüpft sind. Danach wird das neue *TPoint* in *Array* eingefügt.

Wie bereits erwähnt, wäre es besser, *TPoint* als stream-fähig zu definieren. Wenn dies der Fall ist und wenn die *TPoint*-Stream-Fähigkeit in einer anderen Datei implementiert ist, können Sie den Linker anweisen, die Unterstützung der Stream-Fähigkeit für das Objekt *TPoint* dazuzubinden. Wenn das Registrierungsobjekt *TStreamableClass RegPoint* genannt wurde, kann die Stream-Fähigkeit mit dem Makro *__link* dazugebunden werden:

```
__link(RegPoint)
```

Dadurch wird eine externe Referenz auf das Objekt *RegPoint*, das vom Typ *TStreamableClass* ist, erstellt. Da das Registrierungsobjekt einen Zeiger auf die statische Elementfunktion *build* hat, wird die Funktion dazugebunden. Das wiederum bewirkt, daß der Build-Konstruktor und die virtuellen *build-*, *read-* und *write*-Funktionen dazugebunden werden.

III. Windows programmieren

16. Speicherverwaltung

Während der Programmierarbeit müssen Sie sich normalerweise nicht mit der Verwaltung des Speichers durch Windows befassen, da dies – zumindest größtenteils – von Borland C++ erledigt wird.

Verwenden Sie einfach die Routinen new und delete (oder malloc und free) zur Speicherbelegung für dynamische Variablen, der Compiler sorgt dann dafür, daß die erforderliche Reservierung auf dem globalen Heap von Windows erfolgt.

Trotzdem kann es in bestimmten Fällen nötig werden, sich direkt mit der Speicherverwaltung von Windows zu beschäftigen.

Das vorliegende Kapitel geht auf diese Situationen ein und erläutert, wie der Speicher von Windows reserviert und verwendet wird.

Die Speicherverwaltung von Windows

Bei der Entwicklung von Anwendungen für Windows kann es Situationen geben, in denen der Speicherbedarf der Anwendung die Kapazität übersteigt, die vom compilierten Programm direkt zur Verfügung gestellt werden kann. Angenommen, Ihre Anwendung enthält eine Funktion, die eine Bitmap-Datei von der Festplatte lädt und sie auf dem Bildschirm angezeigt.

Das Programm benötigt in diesem Fall einen Platz zum Speichern der Bitmap-Daten, bevor diese an Windows zur Anzeige übergeben werden. Eine Bitmap-Datei kann aber ziemlich groß sein – zu groß, um in einer C++-Variablen Platz zu finden. Zur Lösung dieses Problems verfügt Windows über Funktionen für die Belegung und Manipulation von Speicher, der außerhalb des dem Programm eigenen Speicherbereichs liegt.

Windows ist ein multitasking-fähiges System, es können also mehrere Anwendungen gleichzeitig denselben Speicher benutzen. Der Speicher wird von Windows deshalb so verwaltet, daß er von jeder Anwendung immer optimal genutzt werden kann.

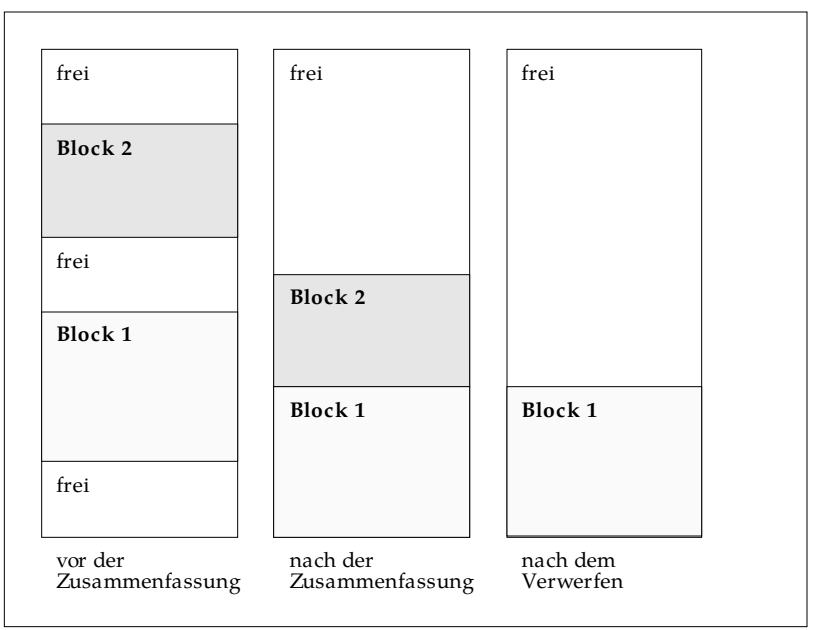
Das vorliegende Kapitel geht ausführlich auf die verschiedenen Speicherarten von Windows ein und erläutert die Mechanismen, die die Erstellung und den Zugriff auf Windows-Speicher innerhalb der Anwendung ermöglichen.

Die Verwaltung des Speichers durch Windows

Bei den meisten auf DOS basierenden Speicherverwaltungssystemen sind die belegten Speicherblöcke fest mit einer bestimmten Adresse verbunden. Dagegen können bestimmte Speicherblöcke in Windows auch *verschoben* und *verworfen* werden. Ein als verschiebbar reservierter Speicherblock hat keine feste Adresse und kann von Windows bei Bedarf verlagert werden. Zum Beispiel können zwei nicht-zusammenhängende Blöcke des reservierten Speichers zusammengeführt werden, um einen größeren freien Speicherblock zu erzeugen.

Bei der Reservierung kann Windows-Speicher auch mit dem Attribut »discardable« versehen werden. Diese Speicherblöcke können von Windows verschoben, aber auch durch die Änderung der Blocklänge auf 0 verworfen werden. Bei dieser »Neureservierung« werden die im Block enthaltenen Daten gelöscht. Die Anwendung muß deshalb den Speicherblock neu reservieren und anschließend mit den korrekten Daten laden. Das Verschieben oder Verwerfen von Speicherblöcken zu einer größeren, zusammenhängenden freien Speichereinheit wird als *Zusammenfassung* bezeichnet.

Abbildung 16.1 zeigt ein Beispiel für eine Zusammenfassung.



Die linke Säule enthält drei nicht-zusammenhängende Blöcke freien Speichers, die durch die Blöcke 1 und 2 voneinander getrennt sind. Nehmen wir an, es wird Speicher angefordert, der größer als jeder einzelne der drei freien Speicherblöcke sein soll. Sind Block 1 und 2 verschiebbar, können die drei

Blöcke zusammengefaßt werden. Daraus ergibt sich ein größerer zusammenhängender freier Speicherblock (mittlere Säule). Ist der Bedarf an freiem Speicher noch höher und ist Block 2 verwerfbar, kann er (durch Änderung der Blocklänge auf 0) aus dem Speicher entfernt werden. Auf diese Weise ergibt sich ein noch größerer zusammenhängender Block mit freiem Speicher (rechte Säule).

Handles und Adressen

Möglicherweise glauben Sie jetzt, die Verwaltung des Windows-Speichers sei sehr schwierig und aufwendig. Glücklicherweise ist dies nicht der Fall. Bei der Reservierung von Windows-Speicher erhält Ihre Anwendung keinen Zeiger, sondern ein Handle für den betreffenden Speicherblock. Das Handle ist nicht die Adresse eines Speicherblocks, sondern ein eindeutiger Name, den die Anwendung in Funktionsaufrufen zur Speicherverwaltung als Referenz auf den Block benutzt. Das Handle ist ungültig, wenn sein Wert 0 ist.

Ohne Adresse ist es aber unmöglich, auf die Daten in einem verschiebbaren Speicherblock zuzugreifen. Um eine Adresse zu erhalten, muß der Block gesperrt werden. Das Sperren eines Blocks liefert einen Zeiger auf den Anfang des Blocks. Der Zeiger behält seine Gültigkeit, solange der Block gesperrt ist. Dies gilt auch für Blöcke, die verschoben oder verworfen werden können.

Real Mode

Im Real Mode muß die Sperre vom Block entfernt werden, nachdem der Zugriff auf die Daten erfolgt ist. Wird dies vergessen, kann Windows den Block nicht mehr verschieben oder verwerfen, die Speicherverwaltung wird somit ineffizient. Nach dem Aufheben der Sperre ist der Zeiger des Blocks nicht mehr gültig.

Protected Mode

Im Protected Mode von Windows (Standard- oder erweiterter 386-Modus) können globale Speicherblöcke bedenkenlos während der gesamten Dauer der Anwendung reserviert und gesperrt bleiben. Negative Auswirkungen auf die Effektivität der Speicherverwaltung und die Ausführungsgeschwindigkeit treten nicht auf.

Reservieren Sie die Speicherblöcke als verschiebbar. Windows kann sie dann im Speicher nach Bedarf verschieben, ihre Adressen bleiben aber trotzdem gültig.

Lokaler und globaler Speicher

Windows ermöglicht es Ihrer Anwendung, Speicherblöcke zu erzeugen und zu benutzen. Der Speicher dafür kann aus zwei Quellen kommen: dem lokalen Heap und dem globalen Heap. Beim globalen Heap handelt es sich um den

Speicher, der frei (nicht belegt) ist und von allen Anwendungen genutzt werden kann. Windows, Windows-Anwendungen und von Windows-Anwendungen belegte globale Speicherblöcke befinden sich alle im globalen Heap. Der Speicher im lokalen Heap kann nur von Ihrer Anwendung genutzt werden.

Bei jeder Speicherreservierung muß angegeben werden, ob lokaler oder globaler Speicher genutzt werden soll. Für die Wahl sind mehrere Faktoren ausschlaggebend. Die Vor- und Nachteile der verschiedenen Speicherarten werden im folgenden kurz aufgelistet:

Lokaler Speicher

- Schnellerer Zugriff.
- Begrenzte Größe (unter 64 KByte).

Globaler Speicher

- Blöcke sind an 32-Byte-Grenzen ausgerichtet.
- Windows reserviert zwanzig Bytes pro Block für den Overhead der Speicherverwaltung.
- Systemweite Begrenzung für Handles.
- Langsamerer Zugriff.
- Mehr Speicher verfügbar.

Der entscheidende Faktor für Ihre Wahl ist normalerweise die Größe des Speichers, der zur Ausführung der betreffenden Aufgabe benötigt wird. Ist mehr als 1 KByte erforderlich, belegen Sie Speicher auf dem globalen Heap. Werden dagegen kleinere Blöcke für einen kurzen Zeitraum benötigt, sollte der Speicher des lokalen Heap verwendet werden.

Der lokale Heap

Auf den Speicher des lokalen Heap kann nur eine bestimmte Instanz Ihrer Anwendung zugreifen. Normalerweise wird in Windows der lokale Heap im Datensegment der Anwendung eingerichtet. Das Datensegment enthält außerdem den Stack und die globalen Variablen der Anwendung. Der übrige Speicher im Datensegment (bis zu 64 KByte) kann als lokaler Heap genutzt werden.

Windows richtet den lokalen Heap für eine Anwendung nicht automatisch ein. Sie können die Größe des lokalen Heap während des Linkens festlegen, indem Sie eine *HEAPSIZE*-Anweisung in Ihre Moduldefinitionsdatei (.DEF) aufnehmen. Sie können stattdessen aber auch die vorgegebene Größe von TLINK akzeptieren (4096 Bytes).

Die maximale Größe, die für den lokalen Heap angegeben werden kann, hängt auch davon ab, ob das Datensegment fest oder verschiebbar ist.

- Ist es fest, kann nur ein lokaler Block belegt werden, der die Größe des Heap, die beim Compilieren und Linken angegeben wurde, nicht überschreitet.

- Ist das Datensegment verschiebbar und wird mehr Speicher angefordert, als für den lokalen Heap festgelegt wurde, versucht Windows, diesen Bedarf durch die Reservierung zusätzlichen Speichers (bis zur 64 KByte-Grenze) zu erfüllen. Windows kann dazu das Datensegment verschieben, wobei allerdings alle far-Zeiger auf lokale Daten ungültig werden.

Lokalen Speicher reservieren und verwenden

Die Windows-Funktion *LocalAlloc* reserviert einen Speicherblock von der im Funktionsaufruf festgelegten Art und Größe. Es muß angegeben werden, ob der Block mit dem Attribut »fixed«, »moveable« oder »moveable« und »discardable« versehen werden soll (ein Block, der verworfen werden kann, muß auch verschoben werden können). Dafür stehen folgende Optionen zur Verfügung, die im *Flags*-Parameter des Funktionsaufrufs angegeben werden müssen:

LMEM_FIXED erzeugt einen festen Speicherblock.

LMEM_MOVEABLE erzeugt einen verschiebbaren Speicherblock.

LMEM_MOVEABLE | LMEM_DISCARDABLE erzeugt einen Speicherblock, der verschoben und verworfen werden kann.

Wird im lokalen Heap ein neuer Block belegt, können hierfür andere Blöcke verschoben oder verworfen werden. Um das Verwerfen zu verhindern, können Sie zum *Flags*-Parameter *LMEM_NODISCARD* hinzufügen. Mit *LMEM_NOCOMPACT* können Sie verhindern, daß andere Blöcke verschoben oder verworfen werden.

War die Reservierung erfolgreich, liefert die Funktion *LocalAlloc* ein Handle für den Speicherblock. Andernfalls ist das Ergebnis 0. Bei einer erfolgreichen Reservierung hat der Speicherblock mindestens die geforderte Größe. Mit der Funktion *LocalSize* kann seine tatsächliche Größe ermittelt werden. Die maximale Größe für einen lokalen Block ist von der Größe des lokalen Heap abhängig.

Sie verfügen nun zwar über ein Handle für den Block, können aber trotzdem noch nicht auf seine Daten zugreifen. Der Block muß zuerst gesperrt werden, um seine Adresse zu erhalten. Dies geschieht durch einem Aufruf der Funktion *LocalLock*. *LocalLock* fixiert den Block zeitweise an einer bestimmten Adresse im lokalen Heap und liefert einen Zeiger auf den Block.

Diese Adresse bleibt solange gültig, bis die Sperre des Blocks mit der Funktion *LocalUnlock* aufgehoben wird. *LocalLock* inkrementiert den Sperrenzähler des Blocks. Der Sperrenzähler (auch Referenzzähler genannt) gibt an, wie oft der betreffende Block bereits gesperrt wurde.

Der von der Funktion *LocalLock* zurückgelieferte Wert muß nicht unbedingt ein gültiger Zeiger sein – Sie sollten ihn deshalb überprüfen.

Der Rückgabewert ist *NULL*, wenn das Handle, das ihr übergeben wurde, ungültig ist oder der Block verworfen wurde (dazu muß für den Block das Attribut *LMEM_DISCARDABLE* angegeben worden sein).

Das folgende Programm reserviert einen Block von 256 Bytes im lokalen Speicher.

```
HANDLE TheLocalHandle;
PSTR TheLocalData;
...

TheLocalHandle = LocalAlloc(LMEM_MOVEABLE | LMEM_DISCARDABLE, 256);
if (TheLocalHandle != 0) {
  TheLocalData = LocalLock(TheLocalHandle);
  if (TheLocalData != NULL)
  {
    // Bearbeiten Sie hier die Daten mit Hilfe des Zeigers
    // TheLocalData
    LocalUnlock(TheLocalHandle);
  }
  else  //Block wurde nicht gesperrt;
}
else //Block wurde nicht reserviert;
}
```

Wird ein lokaler Block mit dem Attribut *LMEM_FIXED* reserviert, kann er nicht verschoben werden. Es ist deshalb nicht erforderlich, *LocalLock* zum Sperren eines Objekts an einer festen Adresse aufzurufen. Nur in diesem Fall ist das von *LocalAlloc* zurückgelieferte 16-Bit-Handle auch die 16-Bit-Adresse des Speicherblocks.

Lokale Speicherblöcke freigeben und verwerfen

Lokale Speicherblöcke können mit der Funktion *LocalFree* freigegeben werden. Durch die Freigabe eines lokalen Blocks wird dieser wieder zum Bestandteil des verfügbaren lokalen Heap, und das zugehörige Handle wird ungültig. Bei der Beendigung einer Anwendung sollten alle nicht mehr benötigten Speicherblöcke freigegeben werden.Mit der Funktion *LocalDiscard* kann ein lokaler Speicherblock explizit verworfen werden. Beim Verwerfen eines lokalen Blocks wird er auf die Größe 0 reduziert und sein Inhalt vom lokalen Heap entfernt. Sein Handle bleibt jedoch gültig und kann durch einen Aufruf der Funktion *LocalReAlloc* mit dem Handle und einer neuen Größe erneut verwendet werden. Sie können die Verarbeitungszeit etwas verkürzen, wenn Sie Handles verwerfen und wieder reservieren anstatt sie freizugeben und neu zu reservieren.

```
HANDLE LocalHandle;
...
// TheLocalHandle wurde bereits reserviert
LocalDiscard(TheLocalHandle);
// Jetzt wird das Handle wieder benutzt, um einen neuen
// Speicherblock zu belegen
LocalHandle = LocalReAlloc(TheLocalHandle, 256, LMEM_MOVEABLE);
```

Ein lokaler Speicherblock kann nur freigegeben oder verworfen werden, wenn sein Sperrenzähler 0 ist.

Lokale Speicherblöcke neureservieren und ändern

Mit der Funktion *LocalReAlloc* ist es möglich, die Größe eines Blocks zu ändern und gleichzeitig seinen Inhalt beizubehalten. Windows »schneidet« Daten ab, wenn eine geringere Größe angegeben wird. Wird dagegen die Größe erhöht und *LMEM_ZEROINIT* angegeben, wird der zusätzliche Bereich mit Nullen initialisiert. Bei einer Vergrößerung ohne Angabe von *LMEM_ZEROINIT* enthält der zusätzliche Bereich undefinierte Daten. Wie bei *LocalAlloc* können andere lokale Blöcke auch bei einer Neureservierung mit *LocalReAlloc* durch Windows verschoben oder verworfen werden. Durch die Verwendung von *LMEM_NODISCARD* oder *LMEM_NOCOMPACT* kann das Verwerfen oder Verschieben anderer Blöcke während der Neureservierung verhindert werden.

Mit *LocalReAlloc* können auch die Attribute des Blocks von *LMEM_MOVEABLE* zu *LMEM_DISCARDABLE* (und umgekehrt) geändert werden. Dazu geben Sie zusätzlich zum neuen Attribut *LMEM_MODIFY* an. *LocalReAlloc* kann aber nicht zusammen mit *LMEM_MODIFY* verwendet werden, um das Attribut *LMEMFIXED* hinzuzufügen oder zu entfernen. Das folgende Programmfragment versieht einen verschiebbaren Block mit dem Attribut »discardable«:

```
HANDLE TheLocalHandle;
...
TheLocalHandle = LocalAlloc(LMEM_MOVEABLE, 256);
TheLocalHandle = LocalReAlloc(TheLocalHandle, 256, LMEM_MODIFY
                              | LMEM_DISCARDABLE);
```

Attribute lokaler Speicherblöcke abfragen

Mit den Funktionen *LocalSize* und *LocalFlags* können Informationen über lokale Speicherblöcke abgerufen werden. *LocalSize* liefert die Größe des lokalen Blocks zurück.

Da bei der Speicherreservierung möglicherweise ein Block erzeugt wird, der die angeforderte Größe überschreitet, können Sie mit *LocalSize* seine tatsächliche Größe wie folgt feststellen:

```
HANDLE TheLocalHandle;
unsigned int BlockSize;

TheLocalHandle = LocalAlloc(LMEM_MOVEABLE, 256);
if (TheLocalHandle != 0)
  BlockSize = LocalSize(TheLocalHandle);
```

Mit *LocalFlags* läßt sich der Sperrenzähler des Speicherblocks abfragen.

Außerdem kann mit dieser Funktion ermittelt werden, ob ein Block verwerfbar ist und ob er verworfen wurde.

```
HANDLE TheLocalHandle;
unsigned int Flags;
unsigned int LockCount;

// Der Block muß bereits reserviert sein
Flags = LocalFlags(TheLocalHandle);
LockCount = Flags & LMEM_LOCKCOUNT;
```

Mit der Funktion *LocalCompact* kann die Größe des größten freien Speicherblocks im lokalen Heap ermittelt werden. Dazu wird als Parameter 0 angegeben. Wenn Sie als Parameter einen Wert ungleich 0 angeben, faßt *LocalCompact* Speicher zusammen und verwirft ihn (falls möglich), um die freien, zusammenhängenden Bytes im lokalen Heap zur Verfügung zu stellen.

Hinweise für die Programmierung

Speicherblöcke, die verschoben oder verworfen werden können, sollten möglichst sofort nach ihrer Benutzung entsperrt werden. Wird die Sperre für diese Blöcke nicht aufgehoben, kann Windows sie nicht verschieben, um für erneute Reservierungen Platz zu schaffen. Außerdem sollten Sie vor dem Programmende alle Speicherblöcke freigeben.

Der globale Heap

Als globaler Heap wird der systemweite Speicher bezeichnet, der von Windows und Ihren Programmen genutzt werden kann. Wieviel Speicher des globalen Heap für Ihre Anwendung reserviert werden kann, hängt von dem Modus ab, in dem Windows läuft. Da der globale Speicher von mehreren Programmen gemeinsam genutzt wird, muß Ihre Anwendung so effizient wie möglich damit umgehen. Andernfalls könnte die Ausführungsgeschwindigkeit im gesamten System darunter leiden.

Reservierung und Verwendung globalen Speichers

Die Funktion *GlobalAlloc* reserviert einen Speicherblock von einer bestimmten Art und Größe, die mit den Argumenten des Funktionsaufrufs festgelegt werden müssen. Wie bei der Zuweisung von lokalem Speicher muß für den Block das Attribut »fixed«, »moveable« oder »moveable« und »discardable« angegeben werden (ein Block, der verworfen werden kann, muß auch verschoben werden

können). Dafür stehen folgende Optionen zur Verfügung, die im *Flags*-Parameter des Funktionsaufrufs angegeben werden müssen:

GMEM_FIXED erzeugt einen festen Speicherblock.

GMEM_MOVEABLE erzeugt einen verschiebbaren Speicherblock.

GMEM_MOVEABLE |GMEM_DISCARDABLE erzeugt einen Speicherblock, der verschoben und verworfen werden kann.

Bei der Reservierung eines neuen Blocks im globalen Heap können andere Blöcke verschoben oder verworfen werden. Um das Verwerfen dieser anderen Blöcke zu verhindern, können Sie zum *Flags*-Parameter *GMEM_NODISCARD* hinzufügen. Mit *GMEM_NOCOMPACT* können Sie verhindern, daß andere Blöcke verschoben oder verworfen werden.

War die Reservierung erfolgreich, liefert die Funktion *GlobalAlloc* ein *Handle* auf den Speicherblock. Andernfalls ist das Ergebnis 0. Der von *GlobalAlloc* zurückgelieferte Wert sollte immer überprüft werden, um festzustellen, ob die Reservierung erfolgreich war. Bei einer erfolgreichen Reservierung hat der Speicherblock mindestens die geforderte Größe. Mit der Funktion *GlobalSize* kann seine tatsächliche Größe ermittelt werden (vgl. den Abschnitt »Globale Speicherblöcke abfragen« in diesem Kapitel). Die maximale Größe für einen globalen Block liegt im Standardmodus bei 1 MByte, erweiterter 386-Modus bei 64 MByte. Um die Adresse eines globalen Speicherblocks zu erhalten, muß er gesperrt werden. Dies geschieht durch einen Aufruf der Funktion *GlobalLock*. Im Real Mode fixiert *GlobalLock* den Block zeitweise an einer bestimmten Adresse im Speicher. Im Protected Mode wird er nicht fixiert, seine Adresse (sein Selector) bleibt aber auch dann gültig, wenn Windows die Blöcke im Speicher verschiebt. *GlobalLock* liefert in allen Modi einen far-Zeiger auf den Block. Dieser bleibt solange gültig, bis die Sperre des Blocks mit der Funktion *GlobalUnlock* aufgehoben wird. Der von *GlobalLock* zurückgelieferte Wert muß nicht unbedingt ein gültiger Zeiger sein – Sie sollten ihn deshalb überprüfen. Der Rückgabewert ist *NULL*, wenn das Handle, das dieser Funktion übergeben wurde, ungültig ist oder der Block verworfen wurde. Das folgende Programm reserviert einen Block im globalen Speicher.

```
unsigned short BigArray[20000];
unsigned short far * TheGlobalData;
HANDLE TheGlobalHandle;

TheGlobalHandle = GlobalAlloc(GMEM_MOVEABLE, sizeof(BigArray));
if (TheGlobalHandle != 0)
{
  TheGlobalData = GlobalLock(TheGlobalHandle);
  if (TheGlobalData != NULL)
  {
    // Bearbeiten Sie hier die Daten mit Hilfe des
    // Zeigers TheGlobalData
    GlobalUnlock(TheGlobalHandle);
  }
```

```
    else
    // Block wurde nicht gesperrt;
  }

  else
  // Block wurde nicht reserviert;
```

Im Gegensatz zu lokalen Speicherblöcken muß ein Block, der mit dem *GMEM_FIXED*-Attribut reserviert wurde, vor einem Zugriff auf seine Daten fixiert werden, da in diesem Fall das von *GlobalAlloc* gelieferte Handle kein Zeiger auf den Block ist.

Weitere Möglichkeiten zum Sperren eines globalen Blocks

Zum Sperren und zum Aufheben der Sperre von globalen Speicherblöcken stehen weitere Funktionen zur Verfügung. *GlobalFix* sperrt den Block im Speicher und verhindert sein Verschieben. Dabei ist es gleichgültig, in welchem Modus sich Windows gerade befindet. Da der Block dadurch fixiert wird, wird der Sperrenzähler um 1 erhöht.

Die Verwendung von *GlobalFix* kann die Zusammenfassung des Speichers durch Windows beeinträchtigen. Setzen Sie *GlobalFix* deshalb nur dann ein, wenn es unbedingt erforderlich ist (dies ist nur bei sehr wenigen Anwendungen der Fall).

GlobalUnfix entsperrt den Block und dekrementiert seinen Sperrenzähler. *GlobalWire* verlagert einen Block in den unteren Speicherbereich und sperrt ihn. Der Sperrenzähler wird dabei um 1 erhöht. Diese Funktion eignet sich besonders für Speicherblöcke, die für einen längere Zeitraum gesperrt bleiben sollen. Da *GlobalWire* einen Block im unteren Speicherbereich plaziert, wird die Zusammenfassung des Speichers nicht beeinträchtigt, solange er fixiert ist. *GlobalUnWire* entfernt die Sperre des Blocks und dekrementiert den Sperrenzähler.

Globale Speicherblöcke freigeben und verwerfen

Globale Speicherblöcke werden auf die gleiche Weise freigegeben und verworfen wie lokale Blöcke. Der einzige Unterschied liegt in den Funktionsnamen: *GlobalFree* und *GlobalDiscard*.

Globale Speicherblöcke neureservieren und ändern

Mit der Funktion *GlobalReAlloc* kann die Größe eines Blocks geändert und gleichzeitig sein Inhalt beibehalten werden. Windows »schneidet« Daten ab, wenn eine geringere Größe angegeben wird. Wird dagegen die Größe erhöht und *GMEM_ZEROINIT* angegeben, wird der zusätzliche Bereich mit Nullen

initialisiert. Bei einer Vergrößerung ohne Angabe von *GMEM_ZEROINIT* enthält der zusätzliche Bereich undefinierte Daten. Wie bei *GlobalAlloc* können vorhandene globale Blöcke auch bei einer Neureservierung mit *GlobalReAlloc* verschoben und verworfen werden. Durch die Verwendung von *GMEM_NODISCARD* oder *GMEM_NOCOMPACT* kann das Verwerfen oder Verschieben von Blöcken während der Neureservierung verhindert werden.

Mit *GlobalReAlloc* können auch die Attribute eines Blocks von *GMEM_MOVEABLE* zu *GMEM_DISCARDABLE* (und umgekehrt) geändert werden.

Wenn Sie für einen Speicherbereich eine neue Größe reservieren, so daß ein Vielfaches einer 64-KByte-Grenze überschritten wird, kann Windows ein neues Handle für diesen Block zurückliefern. Wenn Ihre Anwendung im Standardmodus läuft, trifft das auf die Neureservierung eines Blocks zu, die ein Vielfaches von 65519 Bytes überschreitet. Das alte Handle muß deshalb gespeichert und das von *GlobalReAlloc* zurückgelieferte neue Handle überprüft werden. Ist der Aufruf von *GlobalReAlloc* erfolgreich, müssen Sie das alte Handle durch das neue ersetzen, andernfalls müssen Sie das alte Handle verwenden.

```
HANDLE TheGlobalHandle;
HANDLE OriginalHandle;

// Für den Block wurden ursprünglich 32K reserviert
// Durch Neureservierung wird eine 64K-Grenze überschritten
OriginalHandle = TheGlobalHandle;
TheGlobalHandle = GlobalReAlloc(TheGlobalHandle, 102400,
                                GMEM_MOVEABLE);
if (TheGlobalHandle == 0)
  TheGlobalHandle = OriginalHandle;
```

Globale Speicherblöcke abfragen

Mit den Funktionen *GlobalSize* und *GlobalFlags* können Informationen über globale Speicherblöcke abgerufen werden. *GlobalSize* liefert die Größe des globalen Blocks zurück. Mit *GlobalFlags* läßt sich der Sperrenzähler des Speicherblocks überprüfen. Außerdem kann mit dieser Funktion ermittelt werden, ob ein Block mit dem Attribut »discardable« versehen ist und ob er verworfen wurde. *GlobalFlags* zeigt auch an, ob ein Block als *GMEM_DDESHARE* oder als *GMEM_NOT_BANKED* reserviert wurde. Verwenden Sie *GMEM_DDESHARE* zur gemeinsamen Nutzung von Speicherblöcken durch mehrere Anwendungsinstanzen oder durch eine DLL, auf die mehrere Anwendungsinstanzen zugreifen. *GMEM_NOT_BANKED* kann nur im Real Mode sinnvoll eingesetzt werden.

```
HANDLE TheGlobalHandle;
unsigned int Flags;
...
Flags = GlobalFlags(TheGlobalHandle);
// Ist der Block mit dem Attribut »discardable« versehen?
if (Flags & GMEM_DISCARDABLE)
```

```
  // Ja!
else
  // Nein!
```

Mit der Funktion *GlobalCompact* kann die Größe des größten freien Speicherblocks im globalen Heap ermittelt werden. Dazu wird als Parameter 0 angegeben. *GlobalCompact* führt immer zuerst eine Zusammenfassung durch. Mit dieser Funktion kann auch das Verwerfen aller nicht gesperrten und mit dem Attribut »discardable« markierten Blöcke im System erzwungen werden. Geben Sie dazu -1 als Parameter an.

```
HANDLE TheGlobalHandle;
unsigned long BlockSize;
...

// Größe des größten freien Blocks?
BlockSize = GlobalCompact(0);
// Alle ungesperrten mit dem Attribut »discardable«
// markierten Blöcke verwerfen
GlobalCompact((unsigned long)-1);
```

Globales Verwerfen ändern

Windows stellt mit Hilfe eines LRU-Algorithmus (least-recently-used) fest, welcher verwerfbare Block im Bedarfsfall zuerst verworfen wird. Mit den Funktionen *GobalLRUOldest* und *GobalLRUNewest* können Sie die Position eines Speicherblocks in der Liste mit den zu verwerfenden Blöcken angeben. *GobalLRUOldest* teilt Windows mit, daß der als Parameter übergebene Block als nächstes verworfen werden soll. *GobalLRUNewest* plaziert den angegebenen Block an das Ende der Liste mit den zu verwerfenden Blöcken. Er wird demnach als letzter Block verworfen.

Speicherknappheit

Wenn Windows feststellt, daß es zu viel Zeit für die Zusammenfassung des globalen Heap benötigt, sendet es eine *WM_COMPACTING*-Botschaft an die Hauptfenster der momentan laufenden Anwendungen. Diese Botschaft teilt den Anwendungen mit, daß die Speicherkapazität des Systems bald erschöpft ist, und weist sie an, sofort so viel globalen Speicher wie möglich freizugeben.

Hinweise für die Programmierung

Globale Speicherblöcke sollten möglichst sofort nach ihrer Verwendung entsperrt werden. Wenn die Sperre für diese Blöcke nicht aufgehoben wird, kann

Windows sie nicht verschieben, um für erneute Reservierungen Platz zu schaffen. Vor dem Verlassen des Programms sollten alle Speicherblöcke freigegeben werden.

Aufgrund des Overhead und der Byte-Ausrichtung im globalen Speicher sollten Sie die Reservierung von kleinen globalen Speicherblöcken (unter 128 Bytes) vermeiden. Die Reservierung einer großen Anzahl von globalen Blöcken ist ebenfalls nicht zu empfehlen. Verwenden Sie stattdessen weniger, aber dafür größere Blöcke.

Übergeben Sie nie normale Handles für globale Speicherblöcke an andere Anwendungen, um gemeinsam genutzten Speicher bereitzustellen. Globaler Speicher kann nur dann gemeinsam von mehreren Anwendungen genutzt werden, wenn er mit *GMEM_DDESHARE* reserviert wurde. Sie können zu diesem Zweck auch den dynamischen Datenaustausch und die Zwischenablage verwenden. Näheres finden Sie in Kapitel 18 »Dynamischer Datenaustausch«.

17. Dynamische Linkbibliotheken

Eine dynamische Linkbibliothek (im folgenden kurz DLL genannt) besteht aus einer Bibliothek mit Funktionen, die zur Laufzeit einer Anwendung gelinkt werden.

Anwendungen mit statisch gelinktem Code verfügen über eine eigene Kopie des Codes. Die Funktionen einer DLL können dagegen von allen aufrufenden Anwendungen genutzt werden, d.h. in den Speicher wird nur *eine* Kopie jeder benutzten Funktion einer DLL geladen. Auf diese Weise läßt sich die Größe der .EXE-Dateien der aufrufenden Anwendungen beträchtlich reduzieren.

Eine ObjectWindows-DLL eignet sich vorzüglich zur Aufnahme komplexer Fensterstrukturen, die von verschiedenen Anwendungen gemeinsam genutzt werden können.

Im vorliegenden Kapitel erfahren Sie, wie man eine ObjectWindows-DLL erstellt und verwendet und wie man die dynamische Linkbibliothek von ObjectWindows (OWL.DLL) einsetzt.

DLLs in C++

DLLs können normale Funktionen oder eine Kombination von Funktionen und exportierte C++-Klassen enthalten. Wir wollen uns zunächst mit einer Bibliothek mit normalen Funktionen befassen. Dieser Abschnitt enthält einen Überblick über die DLLs von Borland C++. Es wird beschrieben, wie eine DLL-Funktion mit Hilfe von Borland C++ geschrieben, exportiert und aufgerufen werden kann. In Kapitel 3 »Erstellen einer MS-Windows-Anwendung« des *Borland C++ Benutzerhandbuchs* finden Sie im Abschnitt »Dynamische Linkbibliotheken« weitere Erläuterungen zum Thema DLL.

DLL-Funktionen schreiben

In jeder Windows-DLL müssen die beiden folgenden Funktionen definiert sein: *LibMain* und *WEP* (Windows Exit Procedure).

LibMain und WEP

Die Funktion *LibMain* ist der Haupteinsprung einer Windows-DLL und wird nur einmal beim ersten Laden der Bibliothek aufgerufen. *LibMain* führt die Initialisierung für die DLL aus.

Diese Initialisierung hängt vollständig von der Funktion der jeweiligen DLL ab, umfaßt aber immer folgende Aufgaben:

- Entsperren des Datensegmentes mit *UnlockData*, wenn dieses als MOVEABLE deklariert ist.
- Setzen globaler Variablen für die DLL, falls welche verwendet werden.

Hinweis: Der DLL-Startup-Code C0Dx.OBJ initialisiert den lokalen Heap automatisch.

Folgende Parameter werden an *LibMain* übergeben:

```
int PASCAL LibMain (HANDLE hInstance, WORD wDataSeg,
                    WORD cbHeapSize, LPSTR lpCmdLine)
```

HANDLE, *WORD* und *LPSTR* sind in windows.h definiert.

- *hInstance* ist das Instanzen-Handle der DLL.
- *WORD wDataSeg* ist der Wert des Datensegment-Registers (DS).
- *WORD cbHeapSize* ist die Größe des lokalen Heap, die in der Moduldefinitionsdatei für die DLL angegeben wird.
- *LPSTR lpCmdLine* ist ein far-Zeiger auf die Kommandozeile, die beim Laden der DLL angelegt wird. Diese Zeile ist in dem meisten Fällen leer, da DLLs normalerweise automatisch und ohne Parameter geladen werden. Soll eine dynamische Linkbibliothek explizit geladen werden, dann kann an sie jedoch eine Kommandozeile übergeben werden.

Der Rückgabewert von *LibMain* ist entweder 1 (erfolgreiche Initialisierung) oder 0 (keine Initialisierung). Ist das Ergebnis 0, entfernt Windows die DLL aus dem Speicher.

- *WEP* ist der Austrittspunkt einer DLL. Windows ruft diese Funktion direkt vor dem Entfernen der DLL auf. *WEP* muß nicht unbedingt in einer DLL enthalten sein (weil die Borland C++-Laufzeitbibliotheken über eine Standardfunktion verfügen). Sie kann aber angegeben werden, um »Aufräumarbeiten« zu erledigen, bevor die DLL aus dem Speicher entfernt wird.

Bei Borland C++ muß *WEP* nicht exportiert werden. Der Prototyp der Funktion *WEP* sieht folgendermaßen aus:

```
int FAR PASCAL WEP (int nParameter)
```

int *nParameter* ist entweder WEP_SYSTEMEXIT oder WEP_FREE_DLL. Ersteres bedeutet, daß das gesamte Windows heruntergefahren wird, letzteres bewirkt nur das Entfernen dieser DLL.

WEP sollte 1 zurückliefern, um die erfolgreiche Durchführung anzuzeigen. Derzeit verwendet Windows diesen Rückgabewert allerdings nicht.

DLL-spezifische Funktionen

Die Funktionen, die Sie in Ihre DLL aufnehmen, hängen vom Einsatzzweck der DLL ab. Beim Schreiben dieser Funktionen sollten Sie sich an folgende Richtlinien halten:

- Da die DLL ein anderes Code- und Datensegment als die aufrufende Anwendung verwendet, müssen Aufrufe von DLL-Funktionen far-Aufrufe sein und als Parameter übergebene Zeiger vom Typ far sein.

- Statische Daten in einer DLL sind für alle aufrufenden Anwendungen global. Die globalen Daten einer Anwendung sind also allen anderen Anwendungen zugänglich. Wenn Daten für den Aufrufer einer DLL privat sein sollen, müssen Sie einen anderen Weg einschlagen. Zum Beispiel könnte der Aufrufer den benötigten Speicherplatz bereitstellen und den DLL-Funktionen eine Referenz hierfür übergeben.

DLL-Funktionen exportieren

Nach dem Schreiben der DLL-Funktionen müssen diese exportiert werden, damit sie von einer Anwendung aufgerufen werden können. Dazu sind zwei Schritte erforderlich: die Funktionen müssen als exportierbar compiliert und dann exportiert werden. Dies kann auf verschiedene Weise erfolgen:

- Wenn Sie eine Funktion mit dem Schlüsselwort _export markieren, wird sie als exportierbar compiliert und anschließend exportiert.

- Wenn Sie das Schlüsselwort _export nicht verwenden und beim Compilieren die Option *-WD* angeben, wird die Funktion als exportierbar compiliert. Exportiert wird die Funktion aber nur dann, wenn Sie sie im EXPORTS-Bereich der Moduldefinitionsdatei (.DEF) auflisten.

- Wenn Sie eine Klasse mit dem Schlüsselwort _export deklarieren, wird die gesamte Klasse (Daten und Funktionen) als exportierbar compiliert und exportiert.

DLL-Funktionen aufrufen

Sie können eine DLL-Funktion innerhalb einer Anwendung genau so aufrufen wie jede andere in der Anwendung definierte Funktion. Allerdings müssen Sie dem Linker mitteilen, welche DLL-Funktionen aufgerufen werden. Dazu werden die betreffenden Funktionen *importiert*.

Es gibt zwei Möglichkeiten, eine DLL-Funktion zu importieren:

1. Sie können einen IMPORTS-Bereich in die Moduldefinitionsdatei (.DEF-Datei) der aufrufenden Anwendung einfügen und dort die betreffende DLL-Funktion auflisten.

2. Sie können beim Linken der aufrufenden Anwendung eine Importbibliothek angeben, die Importinformationen für die DLL-Funktion enthält. Verwenden Sie IMPLIB zum Erzeugen der Importbibliothek.

Bei der Ausführung der Anwendung müssen sich die .DLL-Dateien der aufgerufenen DLLs im aktuellen Verzeichnis oder im aktuellen Pfad befinden. Andernfalls läuft Ihre Anwendung nicht.

ObjectWindows-DLLs

ObjectWindows-DLLs exportieren (im Gegensatz zu normalen DLLs) eigene Klassen und von ObjectWindows abgeleitete Klassen. Diese Klassen und die zugehörigen Elementfunktionen können daher von allen Anwendungen genutzt werden, die zu dieser DLL gelinkt werden.

Wenn Sie für eine Gruppe von Anwendungen das gleiche komplexe Verhalten von Fenstern benötigen, können Sie dieses Verhalten in einer ObjectWindows-DLL definieren. Eine derartige DLL könnte z.B. folgendes enthalten:

- Ein komplexes Fenster, das von mehreren Anwendungen verwendet wird.
- Häufig verwendbare Dialoge.
- Häufig verwendbare anwenderdefinierte Steuerelemente.

Das Modul-Objekt

Eine Instanz von *TModule* dient als objektorientierter Ersatz für eine ObjectWindows-DLL. Die Elementfunktionen von *TModule* können zur Fenster- und Speicherverwaltung und zur Fehlerbehandlung eingesetzt werden.

Das folgende Programm erstellt ein Modul-Objekt in der *LibMain*-Funktion einer DLL.

```
PTModule TheModule;

int FAR PASCAL LibMain(HANDLE hInstance, WORD /*wDataSeg*/,
                       WORD cbHeapSize, LPSTR lpCmdLine)
{
  int TheStatus;

  TheModule = new TModule("LibName", hInstance, lpCmdLine);
  TheStatus = TheModule->Status;
  if (TheStatus != 0)
  {
    TheModule->Error(TheStatus);
    delete TheModule;
    TheModule = NULL;
  }
  return (TheStatus == 0);
}
```

In diesem Beispiel wird *LibMain* als Instanz von *TModule* erstellt. Die von Windows erhaltenen Parameter *hInstance* und *lpCmdLine* werden zur Initialisie-

rung von Datenelementen mit identischen Namen an den Konstruktor *TModule* übergeben.

Nachdem das Modul-Objekt erstellt wurde, wird mit Hilfe seines *Status*-Datenelementes festgestellt, ob die Modul-Initialisierung erfolgreich war. Enthält das *Status*-Element einen Wert, der ungleich 0 ist, ist die Initialisierung des Modul-Objektes nicht gelungen. Das Modul-Objekt in *LibMain* sollte dann gelöscht werden. An Windows wird 0 übergeben, um anzuzeigen, daß ein Fehler aufgetreten ist (wie in *LibMain* im Beispiel gezeigt). Wenn die Initialisierung Ihrer DLL nicht erfolgreich durchgeführt werden kann, setzen Sie *Status* auf einen Fehler, den Sie definieren (einen Wert ungleich 0).

In *WEP* wird das Modul-Objekt gelöscht:

```
int FAR PASCAL WEP(int /*bSystemExit*/ {
  delete TheModule;
  return 1;
}
```

Zur Zeit verwendet Windows den Rückgabewert von *WEP* nicht.

Normalerweise erfordert eine DLL weitere Initialisierungs- und Aufräumarbeiten. Sie können dies in der *LibMain*- und der *WEP*-Funktion durchführen, oder (noch besser) eine Klasse von *TModule* ableiten. Im letzteren Fall führen Sie die erforderlichen Initialisierungs- und Aufräumarbeiten im Konstruktor bzw. im Destruktor von *TModule* aus und definieren Datenelemente für die globalen Daten der DLL.

Gemeinsam genutzte Klassen

Sie können den Code zur Definition der Verhaltensweise eines allgemein verwendbaren Dialogs mehreren Anwendungen zur Verfügung stellen, indem Sie eine Dialogklasse in Ihrer DLL definieren. Die Klasse wird dann als gemeinsam genutzte Klasse bezeichnet. Auf diese Weise könnten Sie z.B. einer Gruppe von Grafikanwendungen einen Farbdialog aus der DLL zur Verfügung stellen, der dem Benutzer die Eingabe eines RGB-Werts ermöglicht. Dazu müssen Sie die Klasse aus der DLL exportieren und in die betreffenden Anwendungen importieren.

Deklarieren und Definieren einer gemeinsam genutzten Klasse

Für die erfolgreiche Definition und Deklaration einer gemeinsam genutzten Klasse sind folgende Schritte erforderlich:

- Übergabe eines zusätzlichen *PTModule*-Parameters an Fensterkonstruktoren (dies ist nur in bestimmten Situationen nötig).
- Bedingte Deklarierung der Klasse mit dem Schlüssselwort _export oder huge.
- Verwendung von far-Zeigern und far-Referenzen in den Deklarationen.

Eine gemeinsam genutzte Klasse in einer Include-Datei, die sowohl in der DLL als auch in der Anwendung enthalten ist, muß mit dem Schlüsselwort _export deklariert werden, wenn die DLL compiliert wird, und als huge, wenn

die Anwendung compiliert wird. Dies kann mit Hilfe des Makros *_EXPORT* erfolgen, das – je nachdem, ob eine DLL erstellt oder eine .EXE-Datei mit einer DLL benutzt wird – zu _export oder huge erweitert wird:

```
class __EXPORT TColorControl : public TControl {
protected:
 ...
public:
 ...
};
```

Makro-Erweiterung

Ist *__DLL__* definiert, wird *_EXPORT* zu _export erweitert. Borland C++ definiert *__DLL__* automatisch, wenn Sie beim Compilieren die Option *-WD* oder *-WDE* verwenden oder in der IDE angeben, daß Sie eine DLL aufbauen. Wenn Sie keine DLL erstellen, wird *_EXPORT* zu *_CLASSTYPE* erweitert. *_CLASSTYPE* wiederum ist ein Makro, das zu huge, far oder near erweitert wird.

Damit *_CLASSTYPE* zu huge erweitert wird (dies ist für Klassen erforderlich, die von Anwendungen und DLLs gemeinsam genutzt werden), ist ein drittes Makro nötig: *_CLASSDLL*. Die Definition von *_CLASSDLL* kann auf zwei Arten erfolgen: Entweder in der Kommandozeile des Compilers oder durch die Eingabe in das Editierfeld »Defines« im Dialogfenster mit den Compiler-Optionen der Borland C++ IDE. Ist *_CLASSDLL* einmal definiert, kommuniziert es mit dem Compiler und dem *_CLASSTYPE*-Makro und sorgt dafür, daß die Klassen als gemeinsam genutzte Klassen compiliert werden.

Deklarieren Sie also Ihre gemeinsam genutzten Klassen wie oben gezeigt mit *_EXPORT* und compilieren Sie dann Ihre DLLs mit den entsprechenden Compiler-Direktiven. Die Klassen werden dann exportiert. Bei der Compilierung der .EXE-Datei muß das Makro *_CLASSDLL* definiert sein, damit die Klassen als huge compiliert werden und somit eine gemeinsame Nutzung mit DLLs möglich ist.

Zeiger und typedef

Deklarieren Sie die Zeiger und Referenzen in der Header-Datei Ihrer gemeinsam genutzten Klasse als far-Zeiger. Sie können dazu die Zeiger explizit auf far setzen oder das Makro *_FAR* verwenden. *_FAR* wird nur auf far gesetzt, wenn das Makro *_CLASSDLL* definiert wurde.

ObjectWindows benutzt bei allen Typ-Spezifizierern der ObjectWindows-Klassen das Makro *_FAR*. Der Typ-Spezifizierer für *PTDialog* sieht folgendermaßen aus:

```
typedef TDialog _FAR *PTDialog;
```

ObjectWindows definiert für jede ObjectWindows-Klasse derartige Typ-Spezifizierer. Durch die Verwendung dieser ObjectWindows-Typ-Spezifizierer können Sie die Lesbarkeit Ihres Codes erhöhen.

Wenn Sie einen Zeiger auf eine ObjectWindows-Klasse gebrauchen, verwenden Sie als dessen Typennamen den Namen der Klasse mit einem vorangestellten *P*. Entsprechend verwenden Sie für eine Referenz auf eine Klasse ein *R* gefolgt vom Namen der Klasse. Ein Zeiger auf ein *TWindow*-Objekt ist demnach ein *PTWindow*, eine Referenz auf ein *TDialog*-Objekt ist ein *RTDialog*.

Alle ObjectWindows-Klassen definieren automatisch diese (und weitere) Typ-Spezifizierer, die das *_FAR*-Makro einsetzen. Damit diese Typ-Spezifizierer für Ihre eigenen Klassen definiert werden, geben Sie am Anfang Ihrer Header-Datei für jede Klasse das *_CLASSDEF*-Makro ein. Beispielsweise definieren folgende Programmzeilen *PTMyWindow*, *RTMyWindow* und weitere Typ-Spezifizierer.

```
_CLASSDEF(TMyWindow)
class _EXPORT TMyWindow:public TWindow
{

};
```

Ausführliche Informationen zu den automatisch definierten Datentyp-Spezifizierern finden Sie in der Beschreibung des *_CLASSDEF*-Makros in Kapitel 24 »Weitere Komponenten von ObjectWindows«.

Wenn Ihre gemeinsam genutzten Klassen Zeiger auf Strukturen oder andere Klassen als Datenelemente enthalten, müssen Sie – aufgrund der Art und Weise, in der Borland C++-Compiler Klassen und Strukturen behandelt – möglicherweise referenzierte Strukturen und Klassen mit dem *_CLASSTYPE*-Makro deklarieren:

```
struct _CLASSTYPE TMyStruct
{
};
typedef TMyStruct _FAR * PTMyStruct;

class _EXPORT TMyClass
{
  PTMyStruct ptms;
};
```

In diesem Beispiel wird *TMyStruct* mit *_CLASSTYPE* deklariert, da die huge-Klasse *TMyClass* Zeiger auf *TMyStruct* enthält. (*_EXPORT* impliziert huge bei der Compilierung einer DLL oder einer .EXE-Datei, wenn das Makro *_CLASSTYPE* definiert ist.)

Das folgende Programmbeispiel (es stammt aus der Datei COLORDLG.H) zeigt die Deklaration der gemeinsam genutzten Klasse *TColorDialog* und der von ihr benutzten Klasse *TColorControl*.

```
_CLASSDEF(TColorControl)

class _EXPORT TColorControl : public TControl
{
public:
  COLORREF Color;

  TColorControl(PTWindowsObject AParent,
                int ResourceId, COLORREF AColor,
                PTModule AModule = NULL);

protected:
```

```
    virtual LPSTR GetClassName();
    virtual void GetWindowClass(WNDCLASS _FAR
                                & AWndClass);
    virtual void Paint(HDC, PAINTSTRUCT _FAR & PS);
    virtual WORD Transfer(Pvoid DataPtr, WORD
                          TransferFlag);
    virtual void WMLButtonDown(RTMessage Msg)
      = [WM_FIRST + WM_LBUTTONDOWN];
    virtual void WMLButtonDblClk(RTMessage Msg)
      = [WM_FIRST + WM_LBUTTONDBLCLK];

  public:
    virtual void SetColor(COLORREF AColor);
  };

  _CLASSDEF(TColorDialog)

  class _EXPORT TColorDialog : public TDialog
  {
  protected:
    PTScrollBar ColorBar1;
    PTScrollBar ColorBar2;
    PTScrollBar ColorBar3;
    PTColorControl SelColor;

  public:
    TColorDialog(PTWindowsObject AParent,
                 COLORREF _FAR & TheColor, PTModule TheModule = NULL);

  protected:
    virtual void DefChildProc(RTMessage Msg);
    virtual void SetupWindow();
    virtual void TransferData(WORD TransferFlag);
    virtual void UpdateBars(COLORREF AColor);
  };
```

In diesen Deklarationen wird der Typ-Spezifizierer *RTMessage* aus zwei Gründen verwendet:

- Um die Lesbarkeit des Codes zu verbessern.
- Damit eine far-Referenz verwendet wird, wenn die Klasse als gemeinsam genutzte Klasse eingesetzt wird.

ObjectWindows definiert *RTMessage* folgendermaßen:

```
typedef TMessage _FAR & RTMessage;
```

In der Datei _defs.h sind diverse andere Makros von ObjectWindows definiert, die das Makro *_FAR* verwenden.

Nach der Deklaration der gemeinsam genutzten Klasse können Sie sie in der DLL implementieren.

Eine DLL mit einer gemeinsam genutzten Klasse erstellen

Nach der Deklaration und Definition einer gemeinsam genutzten Klasse sind noch folgende Schritte auszuführen:

- Schreiben der Funktionen *LibMain* und *WEP*.
- Compilieren der DLL.

Sie können die weiter vorne aufgeführten Funktionen *LibMain* und *WEP* in die .CPP-Quelldatei Ihrer DLL kopieren oder selbst erstellte Versionen verwenden. Zum Compilieren der DLL führen Sie einfach die gewohnten Schritte zur Erzeugung von DLLs aus: Wenn Sie den Kommandozeilen-Compiler einsetzen, verwenden Sie die Option *-WDE* oder *-WD*. In der IDE markieren Sie in den Dialogfenstern *Options/Compiler/Entry/Exit code* und *Options/Linker* die entsprechenden DLL-Optionen.

Nachdem Sie Ihre DLL compiliert und gelinkt haben, können Sie mit IMPLIB eine Importbibliothek für Ihre DLL erzeugen. Diese enthält alle exportierten Elementfunktionen Ihrer gemeinsam genutzten Klassen und auch alle gewöhnlichen Funktionen, die Sie exportiert haben.

Weitere Informationen zu diesen Vorgängen finden Sie weiter vorne in diesem Kapitel und in Kapitel 3 »Erstellen einer MS-Windows-Anwendung« im *Borland C++ Benutzerhandbuch* unter »Dynamische Linkbibliotheken«. Eine allgemeine Beschreibung der DLL-Erstellung finden Sie in Ihren Borland C++ Handbüchern.

Eine gemeinsam genutzte Klasse verwenden

Um eine in einer DLL definierte gemeinsam genutzte Klasse zu verwenden, müssen Sie bei der Compilierung der betreffenden Anwendung das Makro *_CLASSDLL* definieren. Dann linken Sie die Importbibliothek hinzu, die Sie für die DLL erstellt haben. Anschließ end können Sie die Klasse so verwenden, als wäre sie direkt in der Anwendung definiert.

Im folgenden Beispiel wird *TColorDialog* ausgeführt, wenn eine Farbe aus dem Menü von *TTestWindow* gewählt wird.

```
void TTestWindow::CMColor(RTMessage) {
  COLORREF TheColor;

  if (GetApplication()->ExecDialog(new TColorDialog(this,
                                   TheColor, GetModule())) ==
                                   IDOK)
}
```

Mit *this* wird ein Zeiger auf das übergeordnete Fenster an den Konstruktor von *TColorDialog* übergeben.

Code, der das Verhalten eines Fensters (z.B. eines komplexen Hilfefensters) oder eines Steuerelements (z.B. eines anwenderdefinierten oder zeichenbaren Steuerelements) definiert, kann ebenfalls gemeinsam genutzt werden. Die hier geschilderte Vorgehensweise trifft für die Definitionen aller gemeinsam genutzten Klassen von ObjectWindows zu.

Sie können eine gemeinsam genutzte Klasse von ObjectWindows als Basisklasse für eine Klasse verwenden, die in einer ObjectWindows-Anwendung oder in einer anderen ObjectWindows-DLL definiert ist.

Eine ObjectWindows-DLL aus einer Nicht-ObjectWindows-Anwendung aufrufen

Bisher sind wir immer davon ausgegangen, daß die Anwendungen, die eine DLL aufrufen, mit ObjectWindows geschrieben wurden. Es ist aber auch möglich, eine ObjectWindows-DLL aus einer Anwendung aufzurufen, die nicht mit ObjectWindows erstellt wurde.

Zu jedem untergeordneten Fenster von ObjectWindows gehört ein übergeordnetes Fensterobjekt. Wird ein untergeordnetes Fenster in einer DLL, das übergeordnete Fenster dagegen in der aufrufenden Anwendung erstellt, muß die Beziehung zwischen über- und untergeordnetem Fenster in der DLL simuliert werden. Dazu wird in der ObjectWindows-DLL ein Fensterobjekt erstellt, das mit dem übergeordneten Fenster verknüpft ist. Das Handle dieses Fensters wird in einem DLL-Aufruf angegeben. ObjectWindows löscht das erzeugte Objekt, wenn das Fenster in der aufrufenden Anwendung freigegeben wird.

Eingehende Botschaften für das übergeordnete Fenster werden zuerst an das zugehörige ObjectWindows-Objekt weitergegeben. Dieses Objekt antwortet nur auf Botschaften, die seine untergeordneten Fenster unterstützen. Alle Botschaften werden an das Fenster in der aufrufenden Anwendung weitergegeben.

Im folgenden Programmbeispiel ist die exportierte *CreateDLLWindow*-Funktion in einer ObjectWindows-DLL enthalten. Sie kann auch von einer .EXE-Datei aufgerufen werden, die nicht mit ObjectWindows erstellt wurde.

```
BOOL_far _export CreateDLLWindow(HWND ParentHWnd)

{
  PTWindowsObject AParentAlias;
  PTWindow TheWindow;
  AParentAlias = DLLHelloLib->GetParentObject(ParentHWnd);
  TheWindow = new TWindow(AParentAlias, "Hello from a DLL!",
                          DLLHelloLib);
  TheWindow->Attr.Style |= WS_POPUPWINDOW | WS_CAPTION |
                           WS_THICKFRAME | WS_MINIMIZEBOX |
                           WS_MAXIMIZEBOX;
  TheWindow->Attr.X = 100;
  TheWindow->Attr.Y = 100;
  TheWindow->Attr.W = 300;
  TheWindow->Attr.H = 300;
  return (DLLHelloLib->MakeWindow(TheWindow) == TheWindow);
}
```

An *CreateDLLWindow* wird das Handle für ein Fenster übergeben, das möglicherweise nicht mit einem ObjectWindows-Fenster übereinstimmt. Der Aufruf von *GetParentObject* liefert ein ObjectWindows-Objekt zurück. Dabei handelt es sich entweder um das *TWindowsObject* von ObjectWindows oder, falls dieses nicht existiert, um ein Ersatzobjekt. Diese *TWindowsObject*-Instanz kann dann so verwendet werden, als wäre sie in der DLL erstellt worden.

Das obige Programmbeispiel erzeugt ein Pop-up-Fenster als untergeordnetes Fenster des angegebenen *ParentHWnd*.

DLLHelloLib, der Modul-Zeiger der DLL, wird als letzter Parameter an den Konstruktor von *TWindow* übergeben, um für das Fenster ein übergeordnetes *TModule* zur Verfügung zu stellen.

ObjectWindows als DLL

Ihre ObjectWindows-Anwendungen kommen mit einer einzigen Kopie der ObjectWindows-Bibliothek aus, wenn Sie sie dynamisch zur DLL von ObjectWindows linken. Dazu sind folgende Schritte erforderlich:

- Definieren Sie beim Compilieren das Makro *_CLASSDLL* (entweder in der Kommandozeile oder in der IDE).
- Wenn Sie nicht das Speichermodel *large* verwenden, müssen alle Zeiger und Referenzen in Klassen-Datenelementen, Parameterlisten und Rückgabewerte das _FAR-Makro verwenden, um Adressen vom Typ far zu erzwingen.
- Beim Linken verwenden Sie anstelle der statischen ObjectWindows-Linkbibliothek (OWLWS.LIB, OWLWM.LIB oder OWLWL.LIB) die DLL-Importbibliothek von ObjectWindow (OWL.LIB).

18. Dynamischer Datenaustausch

Da unter Windows mehrere Programme gleichzeitig ausgeführt werden können, wäre es von Vorteil, wenn diese Programme zur Laufzeit Daten gemeinsam nutzen könnten. Windows unterstützt den Datentransfer zwischen den Anwendungen und stellt dazu eine Zwischenablage, ein Protokoll für den dynamischen Datenaustausch (DDE) und die Möglichkeit zur Definition neuer Windows-Botschaften zur Verfügung.

Die Zwischenablage von Windows

In der Zwischenablage können Daten in verschiedenen Formaten gespeichert werden. Jede Windows-Anwendung, die Daten in die Zwischenablage plaziert, kann dazu ein vordefiniertes Format wählen oder ein eigenes Format definieren. Lädt eine Anwendung Daten aus der Zwischenablage, kann sie durch eine Abfrage das Format der gespeicherten Daten feststellen. Dieselben Daten können in unterschiedlichen Formaten in der Zwischenablage gespeichert werden, um der aufrufenden Anwendung die Wahl zwischen verschiedenen Formaten zu ermöglichen.

Formate für die Zwischenablage

In Aufrufen von Zwischenablage-Funktionen werden die vordefinierten Windows-Formate durch Konstanten mit dem Präfix *CF_* angegeben. In Tabelle 18.1 sind die vordefinierten Formate aufgelistet.

Es handelt sich hier um Standardformate, die von vielen Windows-Anwendungen unterstützt werden. Es besteht die Möglichkeit, spezielle oder selbstdefinierte Formate zu registrieren. Daten, die in einem selbstdefinierten Format in der Zwischenablage gespeichert werden, können nur von Anwendungen geladen werden, die dieses Format kennen.

Jede Anwendung, die ein neues Format definiert, muß dieses Format durch einen Aufruf der Funktion *RegisterClipboardFormat* registrieren. Der erste Aufruf von *RegisterClipboardFormat* in einer Windows-Arbeitssitzung liefert einen

neuen, eindeutigen Formatbezeichner für die Zwischenablage entsprechend den *CF_*-Konstanten. Alle Anwendungen, die zu einem späteren Zeitpunkt ein Format mit identischem Namen registrieren, verwenden denselben Formatbezeichner für dieses neue Datenformat.

Tabelle 18.1 Formate für die Zwischenablage

246 **Dynamischer Datenaustausch** 18

CF_TEXT	Nullterminiertes Zeichen-Array.
CF_BITMAP	Bitmap-Format von Windows.
CF_SYLK	Symbolic-Link-Format von Microsoft.
CF_TIFF	Tag-image-file-Format.
CF_METAFILEPICT	Metafile-picture-Format von Windows.

Daten in der Zwischenablage speichern

Zur Speicherung von Daten in der Zwischenablage müssen vier Schritte ausgeführt werden:

1. Öffnen der Zwischenablage (*OpenClipboard*).
2. Leeren der Zwischenablage (*EmptyClipboard*).
3. Speichern der neuen Daten in der Zwischenablage (*SetClipboardData*).
4. Schließen der Zwischenablage (*CloseClipboard*).

Ein Aufruf von *OpenClipboard* benötigt als Argument ein Handle für ein Fenster. Dadurch wird sichergestellt, daß immer nur ein Fenster mit der Zwischenablage arbeiten kann.

Eine Anwendung, die Daten in der Zwischenablage speichern will, muß diese Daten in den globalen Speicher kopieren und in einem Aufruf der Windows-Funktion *SetClipboardData* ein Handle für diese Daten übergeben (vergleichen Sie Kapitel 16 »Speicherverwaltung«). Nach der Übergabe des Handle gehören die Daten zur Zwischenablage, die Anwendung kann also nicht mehr mit ihnen arbeiten. Deshalb ist es wichtig, daß ein Handle für eine *Kopie* der Daten übergeben wird.

Die Funktion *SetClipboardData* benötigt als Argument einen Zwischenablage-Formatbezeichner (zum Beispiel *CF_TEXT*). Die Anwendung kann *SetClipboardData* mehrmals aufrufen und dabei unterschiedliche Formate verwenden.

Das folgende Programm speichert Daten im *CF_TEXT*-Format in der Zwischenablage.

```
BOOL TMyWindow::CopyText(LPSTR TextString); {
  HANDLE StringGlobalHandle;
  LPSTR StringGlobalPtr;

  StringGlobalHandle = GlobalAlloc(GMEM_MOVEABLE, (LONG)
                                   lstrlen(TextString) + 1);
```

```
    if (StringGlobalHandle)
    {
      StringGlobalPtr = GlobalLock(StringGlobalHandle);
      if (StringGlobalPtr)
      {
        lstrcpy(StringGlobalPtr, TextString);
        GlobalUnlock(StringGlobalHandle);
        if (OpenClipboard(HWindow))
        {
          EmptyClipboard();
          SetClipboardData(CF_TEXT, StringGlobalHandle);
          CloseClipboard();
          return TRUE;
        }
        else GlobalFree(StringGlobalHandle);
      }
      else GlobalFree(StringGlobalHandle);
    }
    return FALSE;
}
```

Die Textdaten der Anwendung – sie werden als *TextString*-Argument an *CopyText* übergeben – werden mit *lstcrpy* zu dem globalen Speicherblock kopiert. Das Handle für diesen Block wird anschließend an *TextString* übergeben, somit kann *TextString* von der Anwendung weiter verwendet werden. Nach dem Aufruf kann die Anwendung den Speicherblock nicht mehr verwenden.

Verwenden Sie bei der Reservierung des globalen Speichers mit *GlobalAlloc* nicht das Flag *GMEM_DISCARDABLE*, da die Zwischenablage die Daten nicht wiederherstellen kann.

Daten aus der Zwischenablage laden

Zum Laden von Daten aus der Zwischenablage muß die Anwendung vier Schritte ausführen:

1. Öffnen der Zwischenablage (*OpenClipboard*).

2. Feststellen, welche Formate sich in der Zwischenablage befinden (*IsClipboardFormatAvailable*).

3. Laden der Daten aus der Zwischenablage (*GetClipboardData*).

4. Schließen der Zwischenablage (*CloseClipboard*).

Das Laden von Daten aus der Zwischenablage erfolgt ebenso wie das Speichern mit Hilfe eines Handle für den globalen Speicher. Da die Zwischenablage Besitzer der Daten ist (und bleibt), muß die Anwendung die mit dem Handle verknüpften Daten kopieren. Dieses Handle sollte – außer zum Kopieren des Blocks – nicht direkt verwendet und manipuliert werden.

Die folgende Funktion lädt Textdaten aus der Zwischenablage.

```
int TMyWindow::PasteText(LPSTR TextString, int TextSize)
{
  HANDLE StringGlobalHandle;
  DWORD StringGlobalSize;
  LPSTR StringGlobalPtr;
  int CharsPasted;

  CharsPasted = -1;
  if (OpenClipboard(HWindow))
  {
    if (IsClipboardFormatAvailable(CF_TEXT))
    {
      StringGlobalHandle = GetClipboardData(CF_TEXT);

      if (StringGlobalHandle)
      {
        StringGlobalSize = GlobalSize(StringGlobalHandle);
        StringGlobalPtr = GlobalLock(StringGlobalHandle);

        if (StringGlobalPtr)
        {
          if (TextSize < StringGlobalSize)
            StringGlobalSize = TextSize ;

          _fstrncpy(TextString, StringGlobalPtr,
                    StringGlobalSize);
          TextString[StringGlobalSize] = '\0';
          GlobalUnlock(StringGlobalHandle);
          CharsPasted = lstrlen(TextString);
        }
      }
    }
    CloseClipboard();
  }
  return CharsPasted;
}
```

Eine Anwendung, die mit mehreren Datenformaten arbeiten kann, sollte zunächst das »informativste« Format anfordern.

Enthält die Zwischenablage zum Beispiel ein binäres Format zur Übergabe von Daten, ist es besser, zuerst dieses Format anzufordern, da es beim *CF_TEXT*-Format durch Konvertierung von Zahlen in Text und zurück zu Rundungsfehlern kommen kann.

Verzögerte Datenkonvertierung

Die Ausführungsgeschwindigkeit von Anwendungen, die viele Datenformate unterstützen, würde sich beträchtlich verlangsamen, wenn die Daten bei jeder Speicherung in der Zwischenablage in alle Formate konvertiert werden müßten. Deshalb besteht die Möglichkeit, die Datenkonvertierung zeitlich solange hinauszuzögern, bis eine andere Anwendung Daten im betreffenden Format anfordert.

Übergeben Sie dazu für alle Formate, deren Konvertierung erst später erfolgen soll, im *SetClipboardData*-Aufruf ein Null-Daten-Handle. Eine Anwendung, die eine verzögerte Datenkonvertierung initiiert, muß die zuletzt in der Zwischenablage gespeicherten Daten noch aufbewahren, damit sie später umgewandelt werden können. Fordert eine andere Anwendung Daten in einem Format an, für das eine verzögerte Konvertierung vereinbart wurde, schickt Windows eine *WM_RENDERFORMAT*-Botschaft an die Anwendung, die das verzögerte Format in der Zwischenablage plaziert hat. Das angeforderte Format für die Daten wird im WORD-Argument der *WM_RENDERFORMAT*-Botschaft angegeben. Die Anwendung sollte dann die Daten im angeforderten Format in der Zwischenablage wie oben beschrieben speichern (mit dem Unterschied, daß die Zwischenablage nicht mehr geleert werden muß).

Die Anwendung, die die verzögerte Konvertierung ausgelöst hat, erhält eine *WM_DESTROYCLIPBOARD*-Botschaft, wenn die Zwischenablage von einer anderen Anwendung geleert wird. Sie kann dann die für eine spätere Konvertierung gespeicherten Daten löschen. Wird eine Anwendung mit verzögerter Datenkonvertierung beendet, erhält sie von Windows eine *WM_RENDERALLFORMATS*-Botschaft, damit sie die Datei in allen verzögerten Formaten in der Zwischenablage speichern kann.

Botschaften zwischen Anwendungen übermitteln

Die meisten Botschaften, die ein Fenster oder eine Anwendung empfängt, stammen von Windows oder von einem anderen Fenster derselben Anwendung (vgl. den Abschnitt »Anwenderdefinierte Botschaften« in Kapitel 7 »Überblick über ObjectWindows«). Es besteht aber auch die Möglichkeit, Botschaften von einer Anwendung zu einer anderen zu übermitteln. Dieser Vorgang ist im Protokoll für den Dynamischen Datenaustausch (DDE) geregelt, Windows verfügt aber auch über einen Mechanismus zur Botschaftsübermittlung zwischen Anwendungen. Dieser Mechanismus eignet sich besonders zum Senden von Benachrichtigungsbotschaften und zur Übergabe einfacher Datenwerte. Zur Übermittlung komplexer Daten sollte jedoch das DDE-Protokoll eingesetzt werden.

Angenommen, Sie haben ein Programmpaket entwickelt, das aus einer elektronischen Mailbox-Anwendung, einem Zeitplaner und einem Kalenderprogramm besteht. Die Mailbox-Anwendung enthält spezielle Routinen zum

Abfangen von Benachrichtigungsbotschaften des Netz-Servers. Empfängt sie eine derartige Botschaft, muß sie die beiden anderen Programme davon in Kenntnis setzen. Dazu können Sie eine neue Botschaft mit dem Namen *WM_EMAILARRIVED* definieren. Jede Anwendung, die diese neue Botschaft sendet oder empfängt, muß sie mit der Windows-Funktion *RegisterWindowMessage* registrieren. Verwenden Sie *SendMessage* oder *PostMessage*, um eine Botschaft von einer Anwendung an eine andere zu übermitteln. Soll jedes geöffnete Fenster die Botschaft erhalten, übergeben Sie *-1* als Parameter für das Fenster-Handle. Zur Bestätigung des Erhalts kann die empfangende Anwendung eine weitere anwenderdefinierte Botschaft zurückschicken.

Im *DWORD*-Parameter (*LParam*) darf kein Zeiger auf Daten übergeben werden, da Windows die Anwendung eventuell verlagert und der Zeiger dabei seine Gültigkeit verliert. Die Übergabe eines Handle für den globalen Speicher ist ebenfalls unzulässig, da der betreffende Speicherblock neu reserviert, verschoben oder verworfen werden kann. Zu Problemen kann es auch kommen, wenn die sendende oder empfangende Anwendung das Speicher-Handle nicht freigibt. Übergeben Sie Daten, wenn möglich, immer direkt im *WORD*- oder *DWORD*-Parameter.

Dynamischer Datenaustausch

Der dynamische Datenaustausch (DDE) ist ein Protokoll, in dem die genauen Modalitäten für den Datentransfer zwischen zwei kooperierenden Anwendungen festgelegt sind. Die Benutzung des DDE-Protokolls setzt voraus, daß diese beiden Anwendungen genau wissen, in welcher Weise sie wechselseitig voneinander anwendungsspezifische Daten anfordern müssen. DDE ist in Form von Windows-Botschaften implementiert (sie beginnen alle mit *WM_DDE_*), die von einem Fenster der einen Anwendung zu einem Fenster der anderen Anwendung geschickt werden, und auf diese Weise Daten oder Anfragen übergeben. Wenn Sie das DDE-Protokoll verwenden wollen, muß Ihr Programm die Datei dde.h enthalten.

Das DDE-Protokoll eliminiert außerdem Probleme, die beim Transfer globaler Daten auftreten können. DDE übergibt Textstrings als globale Atome und verwendet den globalen Speicher (er wird mit dem Flag *GMEM_DDESHARE* reserviert) für die Übergabe von Daten. Als Atome werden globale Stringkonstanten bezeichnet, auf die alle laufenden Windows-Anwendungen zugreifen können.

Begriffe

Wenn zwei Anwendungen gegenseitig Daten austauschen, führen sie miteinander eine *Konversation*. Das Fenster, das die Konversation eröffnet, heißt Client-Fenster, das antwortende Fenster der anderen Anwendung wird als Server bezeichnet. Ein einzelnes Fenster kann gleichzeitig nur an einer Konversation

beteiligt sein. Eine Anwendung, die sich an mehreren Konversationen gleichzeitig beteiligen soll, muß mehrere Fenster erzeugen, von denen sich jeweils eines an einer Konversation beteiligt. Falls erforderlich, können diese zusätzlichen Fenster verborgen werden. Das DDE-Protokoll verwendet für die Spezifikation der auszutauschenden Daten ein dreistufiges System. Die oberste Ebene bildet der Anwendungsname, gefolgt von dem Konversationsthema und dem Datenelement-Namen. Der Anwendungsname ist wichtig, da jede Instanz einer Anwendung normalerweise einen speziellen Satz von Themen »versteht«. Ein Textverarbeitungsprogramm könnte zum Beispiel die Dateinamen von Dokumenten als Konversationsthema benutzen. Das angeforderte Datenelement könnte ein spezieller Absatz oder eine bestimmte Schrift sein.

Anwendung und Thema werden zum Aufbau der Konversation benutzt, das Datenelement ist die übertragene Dateneinheit. Die Daten können in jedem Zwischenablage-Format übertragen werden (vergleichen Sie Tabelle 18.1).

Aufbau einer Konversation

Nachdem der Client eine Konversation mit dem Server eröffnet hat, wird diese fortgeführt, bis sie vom Client oder vom Server wieder beendet wird. Die Botschaften zum Eröffnen und zum Beenden einer Konversation lauten *WM_DDE_INITIATE* bzw. *WM_DDE_TERMINATE*.

Um eine Konversation zu beginnen, schickt der Client mit der Funktion *SendMessage* eine *WM_DDE_INITIATE*-Botschaft und verwendet dabei *-1* als Parameter für das Fenster-Handle. Da das DDE-Protokoll asynchron arbeitet, muß nur *WM_DDE_INITIATE* mit *SendMessage* übergeben werden. Nachdem die Konversation mit einem Fenster eröffnet ist, werden weitere DDE-Botschaften mit der Funktion *PostMessage* übertragen. Im Gegensatz zu *SendMessage* muß bei *PostMessage* die sendende Anwendung nicht auf eine Antwort warten. *PostMessage* hat allerdings den Nachteil, daß die Client-Anwendung für das Abrufen erwarteter Botschaften verantwortlich ist.

Auf eine *WM_DDE_INITIATE*-Botschaft können mehrere Fenster der Server-Anwendung antworten. Der Client muß den gewünschten Server auswählen und zu den anderen *WM_DDE_TERMINATE*-Botschaften senden, da immer nur ein Server mit einem Client Daten austauschen kann.

Wenn eine Anwendung eine Konversation mit *WM_DDE_INITIATE* eröffnet, gibt sie den Namen der anderen Anwendung und das Konversationsthema an. Wird als Anwendungsname eine Null angegeben, kann die Konversation mit jeder laufenden Anwendung geführt werden. Desgleichen kann für das Konversationsthema ein Null-Wert angegeben werden, um eine Konversation über jedes beliebige Thema zu ermöglichen.

Die folgende Elementfunktion eröffnet eine Konversation mit einem Fenster. Sie benötigt einen Anwendungsnamen und ein Konversationsthema als Argument. Sie könnten auch *NULL*-Werte als Argumente verwenden.

```
void TClientWindow::InitiateConversation(LPSTR AppName,
                                         LPSTR TopicName)
{
```

```
  WORD AppGlobalAtom, TopicGlobalAtom;

  if (AppName)
    AppGlobalAtom = GlobalAddAtom(AppName);
  else
    AppGlobalAtom = NULL;
  if (TopicName)
    TopicGlobalAtom = GlobalAddAtom(TopicName);
  else
    TopicGlobalAtom = NULL;
  SendMessage(-1, WM_DDE_INITIATE, HWindow,
              MAKELONG (AppGlobalAtom, TopicGlobalAtom));
  if (AppGlobalAtom)
    GlobalDeleteAtom(AppGlobalAtom);
  if (TopicGlobalAtom)
    GlobalDeleteAtom(TopicGlobalAtom);
}
```

In der Funktion *SendMessage* werden globale Atome anstelle von Zeigern übergeben. Die Atome werden nur dann erzeugt, wenn die Strings mit dem Anwendungsnamen bzw. dem Thema nicht *NULL* sind. Die Anwendung muß nach dem Aufruf von *SendMessage* alle erzeugten globalen Atome freigeben. Dies ist unproblematisch, da der Aufruf von *SendMessage* erst dann zurückkehrt, wenn alle Fenster im System die Möglichkeit hatten, auf die *WM_DDE_INITIATE*-Botschaft zu reagieren. Die Richtlinien für die Erzeugung und Freigabe globaler Atome sind bei der Erläuterung der einzelnen DDE-Botschaften aufgeführt. Alle Anwendungsfenster, die den übergebenen Anwendungsnamen und das Thema bearbeiten können, antworten durch das Senden einer *WM_DDE_ACK*-Botschaft.

Zur Beantwortung einer *WM_DDE_INITIATE*-Botschaft kann das Fensterobjekt einer Anwendung eine Botschaftsantwortfunktion definieren, die nach dem Empfang von *WM_DDE_INITIATE* aufgerufen wird. In dieser Elementfunktion muß das Fensterobjekt das angegebene Thema überprüfen und, falls eine Konversation möglich ist, dem Client-Fenster eine entsprechende Bestätigung übermitteln. Ist das Thema ein Null-Wert, hat das Server-Fenster die freie Wahl des Themas. Die Server-Anwendung erzeugt normalerweise ein neues Fenster zur Verwaltung der Konversation (denken Sie daran, daß es pro DDE-Konversation nur ein Fenster geben kann). Bei der Bestätigung muß ein Handle für dieses neue Fenster übergeben werden.

Im folgenden Programm bestätigt ein Server-Fenster namens *TServerWindow* eine *WM_DDE_INITIATE*-Botschaft:

```
void TServerWindow::ACKTopic(HWND ClientHWnd, HWND
                             ConversationHWnd, LPSTR TopicName)
{
  WORD AppGlobalAtom, TopicGlobalAtom;

  AppGlobalAtom = GlobalAddAtom("GetApplication() -> name");
  TopicGlobalAtom = GlobalAddAtom(TopicName);
```

```
    if (!SendMessage(ClientHWnd, WM_DDE_ACK, ConversationHWnd,
        MAKELONG(AppGlobalAtom, TopicGlobalAtom)))
    {
      GlobalDeleteAtom(AppGlobalAtom);
      GlobalDeleteAtom(TopicGlobalAtom);
    }
  }
```

Anstatt die *HWindow*-Instanzvariable des Fensterobjekts *TServerWindow* zu verwenden, wird das Fenster, das die DDE-Konversation als Server verwaltet, im Parameter *ConversationHWnd* übergeben. So kann auch für jedes Thema ein neues Fensterobjekt erzeugt werden. Der Client kann dann wählen, mit welchem Thema die Konversation fortgeführt werden soll. Beachten Sie, daß erneut globale Atome erzeugt und nur dann gelöscht werden, wenn *SendMessage* keinen Erfolg hat. Das Löschen der globalen Atome muß durch den Client vorgenommen werden, nachdem er die *WM_DDE_ACK*-Botschaft erhalten hat.

Die Konversation ist jetzt aufgebaut. Der Server kennt den Client, dessen Fenster-Handle er mit der *WM_DDE_INITIATE*-Botschaft erhalten hat. Umgekehrt kennt der Client den Server, da er dessen Fenster-Handle über die *WM_DDE_ACK*-Botschaft erhalten hat. Diese Fenster-Handles werden in allen folgenden DDE-Botschaften verwendet, die zwischen den Anwendungen ausgetauscht werden.

Beenden einer Konversation

Die Konversation kann sowohl vom Client- als auch vom Server-Fenster durch das Senden einer *WM_DDE_TERMINATE*-Botschaft beendet werden. Normalerweise übernimmt der Client diese Aufgabe, nur beim Beenden seiner eigenen Anwendung schickt der Server die Botschaft. Nach Erhalt von *WM_DDE_TERMINATE* muß das empfangende Fenster eine *WM_DDE_TERMINATE*-Botschaft zurücksenden. Auf DDE-Botschaften, die nach der Verarbeitung einer *WM_DDE_TERMINATE*-Botschaft eingehen, darf nicht mehr geantwortet werden und alle globalen Atome oder Daten müssen gelöscht werden. Nachdem das sendende Fenster die beantwortende *WM_DDE_TERMINATE*-Botschaft erhalten hat, kann es geschlossen werden.

WM_DDE_TERMINATE-Botschaften können auch unter extremen Bedingungen gesendet werden, etwa bei einem schweren Fehler, der die Beendigung einer Anwendung zur Folge hat. Es kann deshalb vorkommen, daß eine *WM_DDE_TERMINATE*-Botschaft nicht mehr beantwortet wird. Zum Schutz gegen derartige Fälle sollte die sendende Anwendung warten, bis sie eine *WM_DDE_TERMINATE*-Botschaft empfängt oder bis eine gewisse Zeit verstrichen ist. Alle DDE-Botschaften, die die *PostMessage*-Funktion verwenden, sollten über einen derartigen Timeout-Schutz verfügen. Vor einem Aufruf der *PostMessage*-Funktion kann auch *IsWindow* aufgerufen werden, um zu überprüfen, ob der Parameter für das Fenster-Handle des Kommunikationspartners gültig ist. Diese Schutzmechanismen sind nicht Bestandteil des DDE-Protokolls, sollten aber

eingebaut werden, um bei schweren Fehlern eine Fortsetzung der Anwendung zu ermöglichen.

Elementfunktionen für den Datenaustausch

Client und Server können auf fünf Arten miteinander kommunizieren:

1. Mit *WM_DDE_REQUEST* - (Client) und *WM_DDE_DATA*-Botschaften (Server) können einzelne Datenelemente übertragen werden.

2. Ein Client kann über die Änderung eines Datenelements mit Hilfe von *WM_DDE_ADVISE*- und *WM_DDE_DATA*-Botschaften informiert werden. Die Daten können anschließend mit den Botschaften *WM_DDE_REQUEST* - und *WM_DDE_DATA* abgerufen werden. Dieser Mechanismus liegt dem sogenannten *Hot Link* zugrunde. Diese fortlaufende Aktualisierung kann mit *WM_DDE_UNADVISE* beendet werden.

3. Die dritte Methode besteht aus einer Kombination von Methode 1 und 2, wobei die Daten von *WM_DDE_DATA*-Botschaften fortlaufend übertragen werden und somit eine *WM_DDE_REQUEST*-Botschaft nicht erforderlich ist.

4. Eine *WM_DDE_POKE*-Botschaft weist den Server an, den Wert eines Datenelements zu ändern.

5. Eine *WM_DDE_EXECUTE*-Botschaft übergibt ein Befehlsmakro an die Server-Anwendung. Dieses Makro wird dann vom Server interpretiert und ausgeführt.

Anfordern eines einzelnen Datenelements

Normalerweise fordert das Client-Fenster vom Server-Fenster Daten an. Mit dem folgenden Programmfragment kann ein einzelnes Datenelement angefordert werden:

```
void TClientWindow::RequestData(WORD DataFormat, LPSTR Item,
                                HWND ServerHWND)
{
  WORD ItemGlobalAtom;

  ItemGlobalAtom = GlobalAddAtom(Item);
  if (!PostMessage(ServerHWND, WM_DDE_REQUEST, HWindow,
        MAKELONG(DataFormat, ItemGlobalAtom)))
    GlobalDeleteAtom(ItemGobalAtom);
}
```

Der Parameter *DataFormat* enthält ein Format der Zwischenablage, zum Beispiel *CF_TEXT*. Kann der Server die Daten in das geforderte Format konvertieren, antwortet er mit einer *WM_DDE_DATA*-Botschaft:

```
#include <dde.h>
#include <mem.h>
...
void TServerWindow::ACKData(LPSTR Item, WORD DataFormat,
                            LPSTR Data, WORD DataSize)
{
  DDEDATA far *LPDataRecord;
  WORD ItemGlobalAtom;
  WORD DataGlobalHandle;

  DataGlobalHandle = GlobalAlloc(GMEM_MOVEABLE | GMEM_DDESHARE,
                                 (LONG)(sizeof(DDEDATA) +
                                 DataSize));
  if (DataGlobalHandle)
  {
    LPDataRecord = (DDEDATA FAR*)GlobalLock(DataGlobalHandle);
    if (!LPDataRecord)
      GlobalFree(DataGlobalHandle);
    else
    {
      LPDataRecord->fAckReq = LPDataRecord->fResponse = TRUE;
      LPDataRecord->fRelease = FALSE;
      LPDataRecord->cfFormat = DataFormat;
      _fmemcpy(LPDataRecord->Value, Data, DataSize);
      GlobalUnlock(DataGlobalHandle);
      ItemGlobalAtom = GlobalAddAtom(Item);
      if (!PostMessage(ClientHWND, WM_DDE_DATA, HWindow,
          MAKELONG(DataGlobalHandle, ItemGlobalHandle)))
      {
        GlobalFree(DataGlobalHandle);
        GlobalDeleteAtom(ItemGlobalAtom);
      }
    }
  }
}
```

Einige Anmerkungen zu diesem Beispiel: Der globale Speicher wird als *GMEM_DDESHARE* reserviert. Dadurch haben sowohl der Client als auch der Server Zugriff auf die durch das Handle bezeichneten Daten. Der Datenblock-Record steht im globalen Speicher vor den eigentlichen Daten. Auf diese Weise können das Format der Daten und einige Bit-Feld-Optionen (*fResponse* und *fAckReq*) angegeben werden, die den Client anweisen, in einer bestimmten Art und Weise zu antworten. *fResponse* bestimmt, daß die Botschaft *WM_DDE_DATA* als Antwort auf eine *WM_DDE_REQUEST*-Botschaft gesendet wird. *fAckReq* weist den empfangenden Client an, für die empfangene *WM_DDE_DATA*-Botschaft eine *WM_DDE_ACK*-Bestätigungsbotschaft zu senden.

Wird als Datenformat *CF_TEXT* gewählt, muß jede Datenzeile im übergebenen Datenzeiger durch das Zeichenpaar Carriage-Return/Line-Feed beendet werden. Dies gilt auch für die letzte Zeile.

Kann der Server die *WM_DDE_REQUEST*-Botschaft nicht verarbeiten (wenn er beispielsweise das Datenelement nicht in das gewünschte Format konvertieren kann), muß er eine negative Bestätigung an den Client zurücksenden. Generell sollten Sie immer zuerst die komplexen Formate anfordern und sich dann zu den einfacheren »vortasten«, bis eine positive Bestätigung eintrifft. Die negative Bestätigung sieht folgendermaßen aus:

```
...
ItemGlobalAtom = GlobalAddAtom(Item);

if (!PostMessage(ClientHWND, WM_DDE_ACK, HWindow /* des Servers */,
    MAKELONG(0, ItemGlobalAtom)))
  GlobalDeleteAtom(ItemGlobalAtom);
...
```

Nach Erhalt der *WM_DDE_DATA*-Botschaft kann der Client die eingehenden Daten verarbeiten. Er muß den *DDEDATA*-Record am Anfang des Datenzeigers überprüfen und, wenn *fAckReq* gesetzt ist, eine positive Bestätigung zurücksenden:

```
...
if (DataRecord->fAckReq)
{
   ItemGlobalAtom = GlobalAddAtom(Item);
   if (!PostMessage(ServerHWND, WM_DDE_ACK, HWindow /*des Client */,
         MAKELONG(0x8000, ItemGlobalAtom)))
   GlobalDeleteAtom(ItemGlobalAtom);
}
...
```

Wenn *fRelease* gesetzt ist, muß der Client das Speicher-Handle freigeben:

```
...
GlobalUnlock(ItemGlobalHandle);
if (DataRecord->fRelease)
  GlobalFree(ItemGlobalHandle);
...
```

In der *WM_DDE_DATA*-Botschaft sollte auch das Flag *fAckReq* oder *fRelease* (oder beide) gesetzt werden. Der globale Speicher wird entweder vom Server nach Erhalt einer Bestätigung oder vom Client freigegeben. Ist keines dieser Flags gesetzt, weiß der Server nicht, wann der globale Speicher freigegeben werden muß, und der Client wird ihn nicht freigeben.

Dieser Teil des DDE-Protokolls erfüllt drei wichtige Aufgaben:

1. Er stellt sicher, daß alle reservierten globalen Speicherobjekte freigegeben werden.

2. Er reserviert die globalen Speicherobjekte in einem Format, das sowohl vom Client als auch vom Server verarbeitet werden kann.

3. Er verhindert Patt-Situationen, in denen beide Anwendungen auf eine Antwort warten.

Kontinuierlicher Datentransfer

Ein Client kann eine kontinuierliche Verbindung – ein Hot Link – einrichten, in der der Server Änderungen an einem Datenelement meldet. Eine derartige Verbindung wird durch das Senden einer *WM_DDE_ADVISE*-Botschaft eingerichtet:

```
void TClientWindow::Advise(LPSTR Item, WORD DataFormat)
{
  DDEADVISE FAR * LPDataRecord;
  WORD ItemGlobalAtom;
  WORD DataGlobalHandle;

  DataGlobalHandle = GlobalAlloc(GMEM_MOVEABLE | GMEM_DDESHARE,
                                 (LONG)(sizeof DDEADVISE));
  if (DataGlobalHandle)
  {
    LPDataRecord = (DDEADVISE FAR *)GlobalLock(DataGlobalHandle);
    if (!LPDataRecord)
      GlobalFree(DataGlobalHandle);
    else
    {
      LPDataRecord->fAckReq = LPDataRecord->fDeferUpd = TRUE;
      LPDataRecord->cfFormat = DataFormat;
      GlobalUnlock(DataGlobalHandle);
      ItemGlobalAtom = GlobalAddAtom(Item);
      if (!PostMessage(ClientHWND, WM_DDE_ADVISE, HWindow,
          MAKELONG(DataGlobalHandle, ItemGlobalHandle)))
      {
        GlobalFree(DataGlobalHandle);
        GlobalDeleteAtom(ItemGlobalAtom);
      }
    }
  }
}
```

Wenn der Server die Daten in das angeforderte Format konvertieren kann, sollte er eine positive *WM_DDE_ACK*-Bestätigung schicken. Kann er das nicht, muß er eine negative *WM_DDE_ACK*-Botschaft senden. Anschließend kann der Client versuchen, die Verbindung mit einem anderen Format der Zwischenablage aufzubauen.

Wie Sie sehen, ist das Flag *FDeferUpd* in *WM_DDE_ADVISE* gesetzt. Das bedeutet, daß der Client bei einer Änderung der Daten nur eine Bestätigung, nicht aber die Daten, erwartet. Alle folgenden *WM_DDE_DATA*-Botschaften des Servers werden ohne Daten gesendet. Dazu ein Beispiel:

```
...
ItemGlobalAtom = GlobalAddAtom(Item);
if (!PostMessage(ClientHWND, WM_DDE_DATA, HWindow /* des Servers */,
```

```
        MAKELONG(0, ItemGlobalAtomö)))
    GlobalDeleteAtom(ItemGlobalAtom);
...
```

Der Client muß die Daten mit einer *WM_DDE_REQUEST*-Botschaft anfordern. Wenn das *fDeferUpd*-Flag nicht gesetzt ist, enthalten alle folgenden *WM_DDE_DATA*-Botschaften des Servers die Daten.

Den Server zum Ändern eines Datenwertes auffordern

Mit der Botschaft *WM_DDE_POKE* kann der Server aufgefordert werden, einen Datenwert zu ändern. Diese Botschaft wird folgendermaßen eingesetzt:

```
void TClientWindow::Poke(LPSTR Item, WORD DataFormat,
                         LPSTR Data, WORD DataSize)
{
  DDEPOKE FAR * LPDataRecord;
  WORD ItemGlobalAtom;
  WORD DataGlobalHandle;

  DataGlobalHandle = GlobalAlloc(GMEM_MOVEABLE | GMEM_DDESHARE,
                                 (LONG)(sizeof(DDEPOKE) +
                                 DataSize));
  if (DataGlobalHandle)
  {
    LPDataRecord = (DDEPOKE FAR *)GlobalLock(DataGlobalHandle);
    if (!LPDataRecord)
      GlobalFree(DataGlobalHandle);
    else
    {
      LPDataRecord->fRelease = FALSE;
      LPDataRecord->cfFormat = DataFormat;
      _fmemcpy(LPDataRecord->Value, Data, DataSize);
      GlobalUnlock(DataGlobalHandle);
      ItemGlobalAtom = GlobalAddAtom(Item);
      if (!PostMessage(ServerHWND, WM_DDE_POKE, HWindow,
          MAKELONG(DataGlobalHandle, ItemGlobalHandle)))
      {
        GlobalFree(DataGlobalHandle);
        GlobalDeleteAtom(ItemGlobalAtom);
      }
    }
  }
}
```

Der Programmtext von *TClientWindow::Poke* weist eine starke Ähnlichkeit mit dem von *TServerWindow::ACKData* auf. Der Unterschied liegt im Adressaten: *WM_DDE_POKE* wird vom Client zum Server gesendet, während *WM_DDE_-*

DATA vom Server zum Client geschickt wird. Kann der Server die *WM_DDE_-POKE*-Botschaft verarbeiten, muß er eine positive, andernfalls eine negative Bestätigung senden. Der Server könnte bei der Verarbeitung der *WM_DDE_POKE*-Botschaft den Namen des Datenelements zum Beispiel folgendermaßen erhalten:

```
...
char ItemName[128];

ItemGlobalAtom = Msg.LP.Hi;
GlobalGetAtomName(ItemGlobalAtom, ItemName, sizeof (ItemName));
...
```

Makrobefehle im Server ausführen

Mit der Botschaft *WM_DDE_EXECUTE* kann der Client einen nullterminierten String an den Server senden, der als Befehlsmakro ausgeführt werden soll. Handelt es sich beispielsweise beim Server um ein Textverarbeitungsprogramm, muß der String dem Format der Makrosprache des Textverarbeitungsprogramms entsprechen. Der Server muß die Ausführung der *WM_DDE_EXECUTE*-Botschaft durch eine positive oder negative Bestätigung zurückmelden. Das folgende Programm zeigt, wie diese Botschaft eingesetzt werden kann:

```
void TClientWindow::Execute(LPSTRCommand)
{
  WORD DataGlobalHandle;
  LPSTR DataGlobalPtr;

  DataGlobalHandle = GlobalAlloc(GMEM_MOVEABLE | GMEM_DDESHARE,
                                 (LONG)_fstrlen(Command));
  if (DataGlobalHandle)
  {
    DataGlobalPtr = GlobalLock(DataGlobalHandle);
    if (!DataGlobalPtr)
      GlobalFree(DataGlobalHandle);
    else
    {
      _fstrcpy(DataGlobalPtr, Command);
      GlobalUnlock(DataGlobalHandle);
      if (!PostMessage(ServerHWND, WM_DDE_EXECUTE, HWindow,
          MAKELONG(0, DataGlobalHandle)))
      {
        GlobalFree(DataGlobalHandle);
      }
    }
  }
}
```

Der gesendete String muß nullterminiert sein und sollte folgende Syntax haben (die eckigen Klammern sind Teil des Befehls, die geschweiften Klammern enthalten optionale Elemente):

```
[macroCall]{[macroCall]}
```

Das Format von *macroCall* lautet:

```
macroName(parameter1 {, parameter2 ...})
```

Dazu ein Beispiel: Verfügt die Makrosprache Ihres Textverarbeitungsprogramms über die Fähigkeit, mit *goto_line*(*lineno*) eine bestimmte Zeilennummer anzuspringen und den aktuellen Absatz mit *select_current_paragraph* auszuwählen, kann mit dem Befehlsstring

```
[goto_line(50)][select_current_paragraph]
```

der Absatz gewählt werden, in dem sich die Zeile 50 befindet. In Anwendungen, die den dynamischen Datenaustausch als Server unterstützen *und* über eine umfangreiche Makrosprache verfügen, sind praktisch unendlich viele Interaktionen möglich.

Das Thema »System«

Alle Anwendungen, die DDE unterstützen, müssen eine spezielles Thema namens »System« verwalten. Clients, die mit Ihrer Anwendung nicht vertraut sind, können mit Hilfe dieses Themas feststellen, wie DDE in der Anwendung implementiert ist.

Das System-Thema sollte folgende Datenelemente unterstützen:

1. *SysItems*, eine Liste mit Datenelementen des System-Themas.

2. *Topics*, eine Liste mit allen momentan unterstützten Themen. Da einige Anwendungen für jede geöffnete Datei ein anderes Thema definieren, kann sich diese Liste ständig ändern.

3. *Formats*, eine Liste mit den Zwischenablage-Formaten, in die konvertiert werden kann.

Die Einträge in diesen Listen müssen durch Tabulatoren getrennt werden; die Listen selbst müssen dem CF_TEXT-Format entsprechen.

19. Die grafische Geräteschnittstelle

In sehr vielen Windows-Anwendungen setzt sich die Anwenderschnittstelle ausschließlich aus Fenstern, Dialogfenstern und Steuerelementen zusammen. Die Fenster von Grafik-Anwendungen jedoch müssen Grafiken unterschiedlichster Natur aufnehmen können, zum Beispiel Linien, Formen, Text und Bitmap-Bilder.

Die Grafikfähigkeit von Windows beruht auf einer Vielzahl von Funktionen, die unter dem Sammelbegriff *grafische Geräteschnittstelle* (*Graphics Device Interface*, abgekürzt *GDI*) zusammengefaßt werden. Sie können sich die GDI als Grafik-Engine vorstellen, die von Windows-Anwendungen für die Anzeige und die Bearbeitung von Grafiken eingesetzt wird. Die GDI-Funktionen machen Ihre Anwendung von dem für die Anzeige verwendeten Ausgabegerät unabhängig. So sind die Funktionen, die Sie für eine Anzeige auf einem EGA-Bildschirm einsetzen, dieselben wie für die Ausgabe auf einem VGA-Bildschirm oder einem PostScript-Drucker.

Die Geräteunabhängigkeit wird durch Gerätetreiber erreicht, welche die GDI-Funktionsaufrufe in Befehle übersetzen, die das angeschlossene Ausgabegerät »versteht«.

Bildschirmkontexte

Anders als bei herkömmlichen Grafikprogrammen unter DOS setzen Windows-Programme nie direkt die Pixel des Bildschirms, sondern führen Änderungen in einem virtuellen Bereich aus, den wir *Bildschirmkontext* nennen. Ein Bildschirmkontext ist eine virtuelle Oberfläche, die Attribute wie Stift, Pinsel, Schrift, Hintergrundfarbe, Textfarbe und aktuelle Position hat.

Wenn Sie zum Zeichnen in einem Bildschirmkontext GDI-Funktionen aufrufen, übersetzt der zum Kontext gehörige Gerätetreiber den Zeichenbefehl in die entsprechenden Befehle. Diese Befehle setzen den Zeichenbefehl so genau wie möglich für das jeweilige Ausgabegerät um. Es spielt keine Rolle, ob das Ausgabegerät ein Monochrom-Bildschirm oder ein 24-Bit-Farbbildschirm ist.

Sie können sich den Bildschirmkontext als Leinwand für ein Ölgemälde vorstellen, wobei das Fenster das Gemälde samt Rahmen darstellt. Anstatt direkt auf das gerahmte Bild zu zeichnen, zeichnen Sie auf eine Leinwand, die erst

später als fertiges Gemälde auf einen Rahmen gezogen wird. Auf genau diese Weise zeichnen Sie auch Grafiken im Bildschirmkontext. Ein Bildschirmkontext beinhaltet *Zeichenwerkzeuge* wie Stift, Pinsel und Schrift.

Ein Bildschirmkontext ist ein von Windows verwaltetes Element, ähnlich wie ein Fensterelement, mit dem Unterschied, daß zum Bildschirmkontext kein korrespondierendes ObjectWindows-Objekt existiert.

Bildschirmkontexte verwalten

Damit Ihr Programm Grafiken in einen Bildschirmkontext zeichnen kann, muß für das gewünschte Fenster ein derartiger Kontext angefordert werden. Da der Bildschirmkontext sehr viel Speicherplatz benötigt, werden während einer Windows-Sitzung nur fünf Bildschirmkontexte gleichzeitig bereitgestellt. Das heißt, daß es nicht für jedes Fenster einen Bildschirmkontext geben kann. Ein Fenster fordert einen Bildschirmkontext nur dann an, wenn dieser benötigt wird, und gibt ihn schnellstmöglich wieder frei.

Wie Sie Bildschirmkontexte anfordern, verwenden und wieder freigeben, wird weiter hinten in diesem Kapitel im Abschnitt »Grafikanzeige in Fenstern« detailliert erklärt.

Der Inhalt des Bildschirmkontexts

Normalerweise werden Sie die meisten Attribute eines Bildschirmkontextes unverändert übernehmen. Trotzdem ist es wichtig zu wissen, was der Bildschirmkontext enthält. Im vorliegenden Abschnitt erläutern wir kurz seine Eigenschaften wie Bitmaps, Farben, Abbildungsmodi, Clipping-Fähigkeiten und Zeichenwerkzeuge. Auf einige dieser Punkte wird an anderen Stellen im Handbuch detaillierter eingegangen.

Bitmap-Grafiken

Die Oberfläche des Bildschirmkontexts wird als *Bitmap* bezeichnet. Bitmaps repräsentieren die Speicherkonfiguration eines bestimmten Ausgabegeräts. Aus diesem Grund sind Bitmaps abhängig von der Art des adressierten Ausgabegeräts. Darin liegt nun ein Problem: Bitmaps, die von einem Gerät gespeichert wurden, müssen nicht mit einem anderen Ausgabegerät kompatibel sein. Die GDI kann diesem Problem jedoch mit Hilfe der sogenannten *geräteunabhängigen Bitmaps* begegnen. Zu den GDI-Funktionen, die Bitmaps erzeugen, gehören *CreateCompatibleDC*, *CreateCompatibleBitmap* und *CreateDIBitmap*. GDI-Funktionen, die Bitmaps manipulieren, sind unter anderem *BitBlt*, *StretchBlt*, *StretchDIBits* und *SetDIBitsToDevice*.

Farbe

Die Farben, die ein Ausgabegerät zum Zeichnen verwendet, stehen in einer *Farbpalette*. Ist die gewünschte Farbe nicht in der Palette enthalten, können

Sie sie hinzufügen. Genauer gesagt, Sie erlauben dem Gerätetreiber, die gewünschte Farbe durch Abwandlung anderer Farben in der Palette zu erzeugen. Farbpaletten werden im Abschnitt »Der Einsatz von Paletten« weiter hinten in diesem Kapitel erläutert.

Abbildungsmodi

Es ist schwierig Zeicheneinheiten auszuwählen, wenn Sie nicht wissen, welches Ausgabegerät für die Anzeige verwendet wird. Die meisten Anwendungen vernachlässigen dieses Problem einfach und gehen davon aus, daß die Standardeinheit (ein Pixel) ausreicht. Manche Anwendungen machen es jedoch unbedingt erforderlich, daß das gewünschte Bild exakt dargestellt wird. Für diese Anforderungen stellt die GDI verschiedene *Abbildungsmodi* zur Verfügung, die zum Teil geräteunabhängig sind.

Jeder Abbildungsmodus hat eine Einheit und eine Koordinatenorientierung. Der Standardabbildungsmodus setzt den Koordinaten-Ursprungspunkt auf die linke obere Ecke des Bildschirmkontexts, wobei die positive X-Achse nach rechts und die positive Y-Achse nach unten zeigt. Jeder Bildschirmkontext verfügt über ein Abbildungsmodus-Attribut, das festlegt, wie die angegebenen Koordinaten interpretiert werden müssen.

Es gibt Fälle, in denen eine Transformation zwischen den virtuellen Koordinaten, die Sie zum Zeichnen verwenden, und den physikalischen Bitmap-Koordinaten, in die der aktuelle Abbildungsmodus die virtuellen Koordinaten transformiert, erfolgen muß. Bei den meisten Anwendungen ist der Ursprungspunkt des Bildschirms dessen obere linke Ecke. Der Ursprung eines Fensters ist dagegen die obere linke Ecke seines Client-Bereichs. Einige Fenster führen einen Bildlauf für ihre Client-Oberfläche durch, so daß sich der Ursprungspunkt nicht mehr im Client-Bereich befindet. Manche GDI-Funktionen arbeiten mit speziellen Koordinatensystemen, wodurch Konvertierungen erforderlich werden. Die GDI verfügt über Funktionen, die die notwendigen Transformationen ausführen: *ScreenToClient*, *ClientToScreen*, *DPToLP* (Gerätepunkte in virtuelle Punkte) und *LPToDP* (virtuelle Punkte in Gerätepunkte).

Clipping-Regionen

Jeder Bildschirmkontext verfügt über eine Clipping-Region. Darunter versteht man den Bereich, innerhalb dessen Darstellungen des Bildschirmkontextes tatsächlich erscheinen können. Dieser Bereich kann ein komplexes Polygon oder eine elliptische Form sein.

Für die meisten Anwendungen reicht der Client-Bereich des Fensters als voreingestellte Clipping-Region aus. Nur Anwendungen, die spezielle optische Effekte produzieren, müssen eventuell die Clipping-Region ändern.

Zeichenwerkzeuge

Der Bildschirmkontext verwendet zum Zeichnen in erster Linie drei Zeichenwerkzeuge: *Stift*, *Pinsel* und *Schrift*. Der Stift wird zum Zeichnen von Linien, Kreisbögen und Linienzügen (Mehrfach-Liniensegmenten) verwendet. Zu den Attributen eines Stifts gehören Farbe, Breite und Stil (zum Beispiel durchgezogen oder gepunktet).

Ein Pinsel wird zum Ausfüllen von umrahmten Flächen wie Rechtecken,

abgerundeten Rechtecken, Ellipsen und Polygonen eingesetzt. GDI-Funktionen, die umrahmte Flächen zeichnen, verwenden den Stift zum Zeichnen der Einrahmung und den Pinsel zum Ausfüllen der Fläche. Es gibt vier Pinselarten: kompakt, schraffiert, bitmap-gemustert und geräteunabhängig bitmap-gemustert. Das Schrift-Zeichenwerkzeug wird für das Zeichnen von Text im Bildschirmkontext eingesetzt. Es legt Höhe, Schriftstärke, Zeichenabstand, Schriftfamilie und -art fest. Alle drei Zeichenwerkzeuge verwenden das Bildschirmkontext-Attribut der Hintergrundfarbe zum Ausfüllen leergebliebener Flächen.

Die Zeichenwerkzeuge werden im folgenden Abschnitt ausführlich beschrieben.

Zeichenwerkzeuge

Der Bildschirmkontext verwaltet die Anzeige von Grafiken auf dem Bildschirm. Um Grafiken auf verschiedene Art und Weise darzustellen, können Sie die Zeichenwerkzeuge ändern. Die Attribute der Zeichenwerkzeuge bestimmen das Aussehen der mit den GDI-Funktionen (z.B. *LineTo*, *Rectangle* oder *TextOut*) erzeugten Grafiken. Stifte regeln die Darstellung von gezeichneten Linien, Pinsel das Aussehen von gefüllten Flächen und Schriften die Textdarstellung.

Zur Zuordnung von Attributen für Zeichenwerkzeuge wählt das Windows-Programm ein virtuelles oder vordefiniertes Zeichenwerkzeug für den Bildschirmkontext aus. Ein virtuelles Zeichenwerkzeug wird von Ihrem Programm durch Belegen der Felder in einer Struktur (*LOGPEN*, *LOGBRUSH* oder *LOGFONT*) erzeugt.

Ein bereits von Windows vordefiniertes Werkzeug bietet die gebräuchlichsten Attribute, einen kompakten schwarzen Stift, einen grauen Pinsel sowie eine Systemschrift, an.

Vordefinierte Werkzeuge

Handles für vordefinierte Zeichenwerkzeuge können mit der GDI-Funktion *GetStockObject* abgerufen werden.

Sehen Sie sich dazu folgendes Beispiel an:

```
HBRUSH TheBrush;
...
   TheBrush = GetStockObject(LTGRAY_BRUSH);
...
```

LTGRAY_BRUSH ist eine Integerkonstante, die in windows.h definiert ist. Die folgende Tabelle enthält die Konstanten aller verfügbaren vordefinierten Werkzeuge:

Tabelle 19.1 Vordefinierte Zeichenwerkzeuge

Pinsel	Stifte	Schriften	Palette
WHITE_BRUSH	*WHITE_PEN*	*OEM_FIXED_FONT*	*DEFAULT_PALETTE*
LTGRAY_BRUSH	*BLACK_PEN*	*ANSI_FIXED_FONT*	
GRAY_BRUSH	*NULL_PEN*	*ANSI_VAR_FONT*	
DKGRAY_BRUSH		*SYSTEM_FONT*	
BLACK_BRUSH		*DEVICE_DEFAULT_FONT*	
NULL_BRUSH		*SYSTEM_FIXED_FONT*	
HOLLOW_BRUSH			

Vordefinierte Zeichenwerkzeuge dürfen – im Gegensatz zu virtuellen – nach Gebrauch *nicht* gelöscht werden.

Virtuelle Zeichenwerkzeuge

Die Strukturen *LOGPEN, LOGBRUSH* und *LOGFONT* für virtuelle Zeichenwerkzeuge enthalten Felder für jedes Werkzeugattribut. *LOGPEN.lopnColor* enthält beispielsweise den Farbwert für den Stift. Jede Struktur definiert ihre eigenen, dem Werkzeugtyp entsprechenden Attribute.

Zur Erzeugung eines virtuellen Zeichenwerkzeugs übergeben Sie eine entsprechende Struktur mit den gewünschten Attributen an eine GDI-Funktion zur Erstellung des Werkzeugs.

Virtuelle Stifte

Mit den GDI-Funktionen *CreatePen* oder *CreatePenIndirect* können Sie virtuelle Stifte erzeugen:

```
ThePen = CreatePen(PS_DOT, 3, RGB(0, 0, 210));
ThePen = CreatePenIndirect(&ALogPen);
```

Die Definition der *LOGPEN*-Struktur sieht folgendermaßen aus:

```
typedef struct tagLOGPEN {
  WORD     lopnStyle;
  POINT    lopnWidth;
  DWORD    lopnColor;
} LOGPEN;
```

Das Feld *lopnStyle* enthält eine Konstante für die Linienart (siehe Abbildung 19.1).

Das Feld für die Breite, *lopnWidth* , enthält einen Punkt, dessen x-Koordinate – ein Integerwert – die Linienbreite in Geräteeinheiten angibt. Auf einem VGA-Bildschirm wird eine Linie mit einer Breite von 1 Pixel gezeichnet, wenn der Wert 0 ist. Der Wert für die y-Koordinate wird ignoriert.

Das Farbfeld *lopnColor* enthält einen *DWORD* mit den Intensitätswerten der Grundfarben Rot, Grün und Blau, die zur Kombination anderer Farben heran-

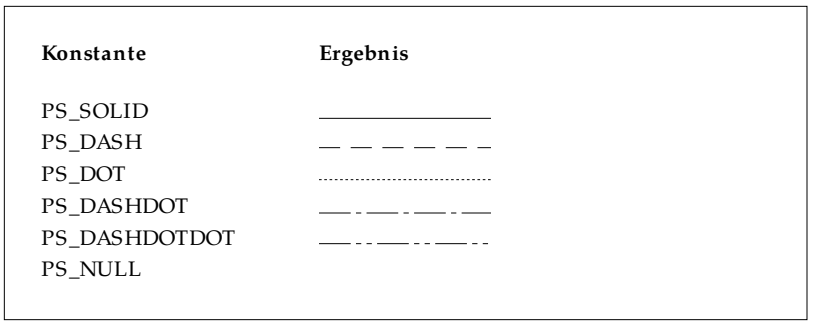


Abbildung 19.1 »Linienarten für Stifte«

gezogen werden. Der Wert in *lopnColor* muß die Form *0x00bbggrr* haben, wobei *bb* der Platzhalter für den Wert Blau, *gg* für Grün und *rr* für Rot ist. Der zulässige Bereich für die Intensität der Grundfarben liegt zwischen 0 und 255, hexadezimal ausgedrückt zwischen 0 und 0xFF.

In der folgenden Tabelle sehen Sie einige Beispiele für Farbwerte:

Tabelle 19.2 Beispiele für RGB-Farbwerte

Wert	Farbe
0x00000000	schwarz
0x00FFFFFF	weiß
0x000000FF	rot
0x0000FF00	grün
0x00FF0000	blau
0x00808080	grau

Alternativ dazu können Sie mit dem *RGB*-Makro Farben erzeugen. Zum Beispiel liefert *RGB(0,0,0)* schwarz, *RGB(255,0,0)* rot, etc.

Virtuelle Pinsel

Mit den GDI-Funktionen *CreateHatchBrush*, *CreatePatternBrush*, *CreateDIBPatternBrush* oder *CreateBrushIndirect* werden virtuelle Pinsel erzeugt, zum Beispiel:

```
TheBrush = CreateHatchBrush(HS_VERTICAL, RGB(0, 255, 0));
TheBrush = CreateBrushIndirect(&ALogBrush);
```

Die Definition der Struktur *LOGBRUSH* sieht folgendermaßen aus:

```
typedef struct tagLOGBRUSH {
  WORD     lbStyle;
  DWORD    lbColor;
  int      lbHatch;
} LOGBRUSH;
```

Das Feld *lbStyle* enthält eine *WORD*-Konstante, die die Pinselart bestimmt:

BS_DIBPATTERN besagt, daß der Musterpinsel von einem geräteunabhängigen Bitmap definiert ist.

BS_HATCHED legt eines der vordefinierten Schraffurmuster fest (vgl. *lbHatch*).

BS_HOLLOW ist ein Pinsel, der mit der Farbe des Hintergrunds zeichnet, also löscht.

BS_PATTERN verwendet das Quadrat mit 8 x 8 Pixel in der linken oberen Ecke eines Bitmap, das aktuell im Speicher steht.

BS_SOLID ist ein kompakter Pinsel.

Das *lbColor*-Feld enthält einen Farbwert, wie er für die *LOGPEN*-Strukturen erforderlich ist. Das Feld wird von Pinseln mit den Stilen *BS_HOLLOW* und *BS_PATTERN* ignoriert.

Das *lbHatch*-Feld enthält eine Integerkonstante, die das Schraffurmuster für Pinsel mit dem Stil BS_*HATCHED* angibt. Wenn der Stil *BS_DIBPATTERN* ist, enthält *lbHatch* ein Handle für das Bitmap. Im Abschnitt »Erstellung eines Pinsels mit Hilfe eines Bitmap« in Kapitel 20 »Details über Ressourcen« können Sie sehen, wie man eine Bitmap-Ressource als Pinsel verwendet.

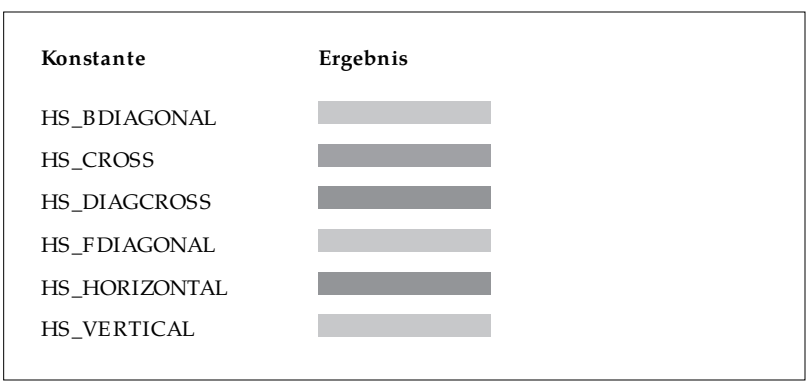


Abbildung 19.2 Schraffurmuster für Pinsel

Virtuelle Schriften

Mit den GDI-Funktionen *CreateFont* oder *CreateFontIndirect* können Sie virtuelle Schriften erzeugen.

Die Definition der *LOGFONT*-Struktur sieht folgendermaßen aus:

```
typedef struct tagLOGFONT {
  int       lfHeight;
  int       lfWidth;
  int       lfEscapement;
  int       lfOrientation;
  int       lfWeight;
  BYTE      lfItalic;
  BYTE      lfUnderline;
  BYTE      lfStrikeOut;
```

```
    BYTE        lfCharSet;
    BYTE        lfOutPrecision;
    BYTE        lfClipPrecision;
    BYTE        lfQuality;
    BYTE        lfPitchAndFamily;
    BYTE        lfFaceName[LF_FACESIZE];
} LOGFONT;
```

Zur Erstellung eines Schrift-Werkzeugs mit *LOGFONT* geben Sie die Attribute der gewünschten Schriften an. Ihr Programm übernimmt diese Informationen allerdings nicht, um damit eine neue Bildschirmschrift zu erzeugen, sondern es bildet die Schriftanforderung auf eine aktuell in der Windows-Sitzung definierte Bildschirmschrift ab.

Das *lfHeight*-Feld legt die gewünschte Höhe der Schrift fest. Bei der Angabe von Null wird eine Standardgröße herangezogen. Ein positiver Wert steht für die Zeichenhöhe in virtuellen Einheiten. Ein negativer Wert wird in einen positiven Wert umgewandelt und ergibt die Zeichenhöhe in virtuellen Einheiten.

Das Feld *lfWidth* gibt die gewünschte Zeichenbreite in Geräteeinheiten an. Wenn der Wert Null ist, wird sie dem Höhen- und Seitenverhältnis des Bildschirms entsprechend berechnet.

Für rotierten Text setzt *lfEscapement* einen Wert in Zehntel Grad, um den der Text gegen den Uhrzeigersinn gedreht wird. *lfOrientation* hat dieselbe Funktion für die Drehung einzelner Zeichen.

Die gewünschte Schriftstärke wird mit *lfWeight* festgelegt. Dazu stehen Ihnen die Konstanten *FW_LIGHT*, *FW_NORMAL*, *FW_BOLD* und *FW_DONTCARE* zur Verfügung. Sie können auch jeden Wert zwischen 0 und 1000 angeben.

Wenn *lfItalic*, *lfUnderline* und *lfStrikeOut* ungleich Null sind, werden die entsprechenden Schriftattribute – kursiv, unterstrichen und durchgestrichen – verwendet.

Das Feld *lfCharSet* fordert bestimmte Zeichensätze wie *ANSI_CHARSET*, *OEM_CHARSET* oder *SYMBOL_CHARSET* an. *OEM_CHARSET* ist systemabhängig.

Das Feld *lfOutPrecision* bestimmt, wie genau die von Windows zur Verfügung gestellte Schrift mit der angeforderten Größe und Position übereinstimmen muß. Voreingestellt ist *OUT_DEFAULT_PRECIS*.

Das Feld *lfClipPrecision* legt fest, wie teilweise sichtbare Zeichen abgeschnitten werden. Voreingestellt ist *CLIP_DEFAULT_PRECIS*.

Das Feld *lfQuality* zeigt an, wie genau die von Windows zur Verfügung gestellte Schrift mit den angeforderten Schriftattributen übereinstimmen soll. Das Feld kann mit *DEFAULT_QUALITY*, *DRAFT_QUALITY* oder *PROOF_QUALITY* belegt werden. Bei *PROOF_QUALITY* werden fette, kursive, unterstrichene und durchgestrichene Schriften künstlich erzeugt, falls sie nicht verfügbar sind.

Das Feld *lfPitchAndFamily* fordert einen Zeichenabstand und eine Schriftfamilie an. Es kann sich aus einer bitweisen ODER-Operation zwischen einer Zeichenabstand- und einer Familienkonstante ergeben (siehe Tabelle 19.3).

lfFaceName ist ein String, der die angeforderte Schriftart festlegt. Wenn der Wert Null ist, wird eine Schriftart verwendet, die auf den Werten in den anderen *LOGFONT*-Feldern basiert.

Tabelle 19.3 Zeichenabstand- und Familienkonstanten für Schriften

Zeichenabstand	Familie
DEFAULT PITCH	*FF_MODERN*
FIXED_PITCH	*FF_ROMAN*
VARIABLE_PITCH	*FF_SCRIPT*
	FF_SWISS
	FF_DECORATIVE
	FF_DONTCARE

Im folgenden finden Sie einige Beispielschriften mit dem zugehörigen Quelltext:

Sample Text

```
void MyWindow::MakeFont()
{
  LOGFONT MyLogFont;

  MyLogFont.lfHeight         = 20;
  MyLogFont.lfWidth          = 0;
  MyLogFont.lfEscapement     = 0;
  MyLogFont.lfOrientation    = 0;
  MyLogFont.lfWeight         = FW_BOLD;
  MyLogFont.lfItalic         = 0;
  MyLogFont.lfUnderline      = 0;
  MyLogFont.lfStrikeOut      = 0;
  MyLogFont.lfCharSet        = ANSI_CHARSET;
  MyLogFont.lfOutPrecision   = OUT_DEFAULT_PRECIS;
  MyLogFont.lfClipPrecision  = CLIP_DEFAULT_PRECIS;
  MyLogFont.lfQuality        = DEFAULT_QUALITY;
  MyLogFont.lfPitchAndFamily = VARIABLE_PITCH | FF_SWISS;
  strcpy(MyLogFont.lfFaceName, "Helv");

  TheFont = CreateFontIndirect(&MyLogFont);
}
```

Sample Text

```
void MyWindow::MakeFont()
{
  LOGFONT MyLogFont;

  MyLogFont.lfHeight         = 10;
  MyLogFont.lfWidth          = 0;
  MyLogFont.lfEscapement     = 0;
  MyLogFont.lfOrientation    = 0;
  MyLogFont.lfWeight         = FW_NORMAL;
```

```
  MyLogFont.lfItalic         = TRUE;
  MyLogFont.lfUnderline      = TRUE;
  MyLogFont.lfStrikeOut      = 0;
  MyLogFont.lfCharSet        = ANSI_CHARSET;
  MyLogFont.lfOutPrecision   = OUT_DEFAULT_PRECIS;
  MyLogFont.lfClipPrecision  = CLIP_DEFAULT_PRECIS;
  MyLogFont.lfQuality        = DEFAULT_QUALITY;
  MyLogFont.lfPitchAndFamily = FIXED_PITCH | FF_DONTCARE;
  strcpy(MyLogFont.lfFaceName, "Courier");

  TheFont = CreateFontIndirect(&MyLogFont);
}
```

Σαμπλε Τεξτ

```
void MyWindow::MakeFont()
{
  LOGFONT MyLogFont;

  MyLogFont.lfHeight         = 20;
  MyLogFont.lfWidth          = 0;
  MyLogFont.lfEscapement     = 0;
  MyLogFont.lfOrientation    = 0;
  MyLogFont.lfWeight         = FW_NORMAL;
  MyLogFont.lfItalic         = 0;
  MyLogFont.lfUnderline      = 0;
  MyLogFont.lfStrikeOut      = 0;
  MyLogFont.lfCharSet        = SYMBOL_CHARSET;
  MyLogFont.lfOutPrecision   = OUT_DEFAULT_PRECIS;
  MyLogFont.lfClipPrecision  = CLIP_DEFAULT_PRECIS;
  MyLogFont.lfQuality        = PROOF_QUALITY;
  MyLogFont.lfPitchAndFamily = VARIABLE_PITCH | FF_ROMAN;
  strcpy(MyLogFont.lfFaceName, "Tms Rmn");

  TheFont = CreateFontIndirect(&MyLogFont);
}
```

Grafikanzeige in Fenstern

Eine Windows-Anwendung sorgt für das Neuzeichnen ihrer Fenster, wenn sie zum ersten Mal angezeigt oder aktualisiert werden müssen, zum Beispiel wenn sie nach der Darstellung als Symbol wiederhergestellt oder in der Größe verändert werden.

Da Windows den Fensterinhalt nicht automatisch neu zeichnet, muß das

Fenster benachrichtigt werden, wenn es sich selbst neu zeichnen soll. Im vorliegenden Abschnitt erfahren Sie, wie in einem Fenster gezeichnet wird, wie der Aktualisierungsmechanismus abläuft und wie Bildschirmkontexte verwendet werden.

Im folgenden werden mit *Grafiken* sowohl Textelemente als auch grafische Elemente bezeichnet, wie beispielsweise Bitmaps und Rechtecke.

Zeichnen in Fenstern

Um Text oder Grafiken in einem Fensterobjekt zeichnen zu können, müssen Sie einen Bildschirmkontext *anfordern*, der nach dem Zeichenvorgang *freigegeben* werden muß. (Während einer Windows-Sitzung stehen maximal fünf Bildschirmkontexte zur Verfügung.) In der Zwischenzeit können Sie das Bildschirmkontext-Handle als Argument in jeder GDI-Funktion einsetzen.

Einen Bildschirmkontext verwalten

Normalerweise werden Sie ein Datenelement im Fensterobjekt definieren, das das Handle für den aktuellen Bildschirmkontext enthält, vergleichbar mit *HWindow*, in dem ein Handle für ein Fenster gespeichert ist:

```
class TMyWindow : public TWindow
{
  HDC TheDC;
  ...
};
```

Wenn Sie für ein Fenster einen Bildschirmkontext anfordern wollen, rufen Sie die Windows-Funktion *GetDC* auf:

```
TheDC = GetDC(HWindow);
```

Danach können Sie Zeichenoperationen für den Bildschirmkontext ausführen und das Bildschirmkontext-Handle an GDI-Funktionen wie z.B. *Rectangle* übergeben:

```
Rectangle(TheDC, 10, 10, 100, 1000);
```

Sobald Sie die Arbeit mit dem Bildschirmkontext beendet haben, geben Sie ihn mit einem Aufruf der Funktion *ReleaseDC* frei:

```
ReleaseDC(HWindow, TheDC);
```

Zwischen zwei aufeinanderfolgenden Aufrufen von *GetDC* sollte immer ein *ReleaseDC*-Aufruf stehen. Andernfalls kann es während des Programmablaufs zu einem Systemfehler kommen, weil der für die Bildschirmkontexte zur Verfügung stehende Speicherplatz nicht mehr ausreicht.

Grafikfunktionen in Fenstern aufrufen

Normalerweise werden Sie GDI-Funktionen aus den Elementfunktionen eines Fensters aufrufen. So ist *TextOut* zum Beispiel eine Funktion, die Text für einen Bildschirmkontext an der angegebenen Stelle zeichnet:

```
void TMyWindow::DrawText
{  TheDC = GetDC(HWindow);
   TextOut(TheDC, 50, 50, "Sample Text", 11);
   ReleaseDC(HWindow, TheDC);
}
```

Fenster aktualisieren

Wenn ein Fenster erneut gezeichnet werden muß, ist es *ungültig* geworden, d.h. seine Anzeige ist nicht mehr korrekt und muß aktualisiert werden. Dies ist der Fall, wenn ein Fenster zum ersten Mal angezeigt wird, wenn es nach der Verkleinerung auf ein Symbol wiederhergestellt wird oder wenn ein anderes Fenster so verschoben wird, daß Teile des dahinterliegenden Fensters sichtbar werden.

In all diesen Fällen sendet Windows eine *WM_PAINT*-Botschaft an die entsprechende Anwendung.

TWindow definiert die Elementfunktion *TWindow::WMPaint* als Antwort auf die *WM_PAINT*-Botschaft, die unter anderem folgendes ausführt:

- Anforderung eines Bildschirmkontexts.
- Aufruf der virtuellen Elementfunktion *Paint*, wobei der Bildschirmkontext als Parameter übergeben wird.
- Freigabe des Bildschirmkontexts.

Die Elementfunktion *Paint* von *TWindow* zeichnet nichts, da *TWindow*-Objekte keine Grafiken beinhalten. Wenn Ihr Fenster Grafiken und Text enthält, definieren Sie eine *Paint*-Elementfunktion, die GDI-Funktionen zum Zeichnen von Grafiken und Text aufruft.

Mit Hilfe des Bildschirmkontexts können Sie neue Zeichenwerkzeuge auswählen, wie beispielsweise Stifte mit verschiedenen Farben oder Pinsel mit verschiedenen Mustern. Sie müssen diese Werkzeuge erneut über die *Paint*-Elementfunktion in den Bildschirmkontext aufnehmen.

Grafik-Strategie

Die *Paint*-Elementfunktion sorgt dafür, daß die aktuellen Fensterinhalte jederzeit neugezeichnet werden können. Dazu gehört auch die erste Anzeige eines Fensters.

Aus diesem Grund muß die *Paint*-Elementfunktion alle »permanenten« Grafiken des Fensters zeichnen können. Zusätzlich sollte sie Grafiken, die seit der ersten Anzeige des Fensters hinzugekommen sind, rekonstruieren können. Dazu muß die *Paint*-Elementfunktion auf die Befehle oder Daten Zugriff haben, mit denen die Grafiken erstellt wurden.

Um dies zu erreichen, haben Sie die Wahl zwischen zwei Möglichkeiten. Die

erste Möglichkeit besteht darin, daß Sie zuerst den grafikerzeugenden Programmtext in einige wenige Elementfunktionen zerlegen und diese dann von der *Paint*-Elementfunktion aus aufrufen, wenn Grafiken erneut erstellt werden müssen.

Der zweite Weg besteht – wie in dem Beispiel in Kapitel 3 »Ein Fenster füllen« ersichtlich – darin, die den Grafikinhalt eines Fensters betreffenden Daten in einem Feld des entsprechenden Fensterobjekts zu speichern. Zu diesen Daten gehören beispielsweise Koordinaten, Formeln und Bitmaps. Rufen Sie dann in der *Paint*-Elementfunktion die Grafikroutinen, die für die Transformierung der Daten in Grafiken erforderlich sind, erneut auf. Mit dieser Strategie und der Fähigkeit von Objekten, die eigenen Daten und Funktionen speichern zu können, steht der Produktion von »lebendigen« Grafikanwendungen nichts mehr im Wege.

Zeichenwerkzeuge verwenden

Der Bildschirmkontext erlaubt Ihnen nicht nur das Zeichnen in einem Fenster, sondern er stellt auch die dafür notwendigen Werkzeuge zur Verfügung: Stifte, Pinsel, Schriften und Paletten. Wenn Sie im Bildschirmkontext eine Linie zeichnen, wird die Linie gemäß der Attribute des aktuellen Stifts erzeugt. Dazu gehören Farbe, Art (durchgezogen, gepunktet, etc.) und Stärke. Das Ausfüllen eines Bereichs richtet sich nach den Attributen des aktuellen Pinsels, also nach seinem Füllmuster und seiner Farbe. Wenn Sie Text in einem Bildschirmkontext zeichnen, erscheint er mit der Schrift (Modern, Roman, Swiss, etc.), der Größe und dem Stil (kursiv, fett, etc.) des aktuellen Schrift-Werkzeugs. Eine Palette enthält eine Kollektion der aktuell verfügbaren Farben.

Für einen Bildschirmkontext gibt es jeweils ein Zeichenwerkzeug von jedem Typ. Ein neu zugewiesener Bildschirmkontext verwendet die voreingestellten Werkzeuge: einen dünnen schwarzen Stift, einen kompakten schwarzen Pinsel, eine Systemschrift und eine Standardpalette. Wenn Ihnen diese Werkzeuge ausreichen, können Sie sie unverändert übernehmen.

Falls Sie die voreingestellten Werkzeuge ändern möchten, müssen Sie ein neues Werkzeugelement erstellen und für den Bildschirmkontext *auswählen*. Wenn Sie zum Beispiel einen neuen Stift wählen, wird der alte Stift deaktiviert. Sie sollten die vorhergehenden Werkzeuge speichern und wieder auswählen, wenn Sie die Arbeit mit dem neuen Werkzeug beendet haben:

```
{
  HPEN NewPen, OldPen;
  HDC TheDC;
  // erzeugt einen Stift mit der Breite 10
  NewPen = CreatePen(PS_SOLID, 10, RGB(0,0,0));
  TheDC = GetDC(AWindow->HWindow);
  OldPen = SelectObject(TheDC, NewPen);

  // hier wird gezeichnet
  SelectObject(TheDC, OldPen);
```

```
    ReleaseDC(AWindow->HWindow, TheDC);
    DeleteObject(NewPen);
}
```

Wie dieses Beispiel deutlich macht, müssen neue Zeichenwerkzeuge erstellt und dann gelöscht werden. Zeichenwerkzeuge sind wie Bildschirmkontexte als Elemente im Windows-Speicher abgelegt. Werden die Werkzeuge nicht gelöscht, belegen sie unnötig Speicher und es kommt schließlich zu Fehlern. Ebenso wie bei den Bildschirmkontexten sollten Sie Handles für Zeichenwerkzeuge in Variablen der Typen *HPEN*, *HBRUSH*, *HFONT* und *HPALETTE* speichern. Die Windows-Funktion *DeleteObject* entfernt Zeichenwerkzeuge aus dem Windows-Speicher. *Löschen* Sie aber *niemals* die Zeichenwerkzeuge, die momentan für einen Bildschirmkontext ausgewählt sind.

Obwohl ein Bildschirmkontext immer nur ein Zeichenwerkzeug desselben Typs enthalten kann, können Sie ein »Lager« mit verfügbaren Werkzeugen anlegen. Wichtig ist nur, daß Sie alle Werkzeuge wieder löschen, wenn Ihre Anwendung beendet wird. Sie können dabei so vorgehen, wie es in dem Beispiel in Kapitel 3 »Ein Fenster füllen« gezeigt wird: Definieren Sie ein Datenelement mit dem Namen *ThePen* für Ihr Fensterobjekt, um darin ein Handle für das aktuelle Stift-Werkzeug zu speichern. Wenn der Anwender eine neue Stiftart auswählt, wird ein neues Stift-Werkzeug erstellt und das alte gelöscht. Der zuletzt aktive Stift wird im Destruktor des Hauptfensters gelöscht. Die voreingestellten Werkzeuge, die ein neu abgerufener Bildschirmkontext enthält, dürfen Sie nicht löschen.

Es gibt zwei Möglichkeiten zum Anfordern eines Handle für ein neues Zeichenwerkzeug. Der einfachste Weg ist die Verwendung eines bereits existierenden, *vordefinierten* Werkzeugs. Die vordefinierten Werkzeuge finden Sie in Tabelle 19.1.

Wenn Sie ein vordefiniertes Werkzeug für den Bildschirmkontext auswählen wollen, verwenden Sie die Elementfunktionen *GetStockPen*, *GetStockBrush*, *GetStockFont* und *GetStockPalette*. Sehen Sie sich das folgende Beispiel an:

```
ThePen = GetStockObject(BLACK_PEN);
```

Wichtig: Löschen Sie die vordefinierten Werkzeuge nicht aus dem Windows-Speicher.

Es kann vorkommen, daß es kein vordefiniertes Werkzeug gibt, das über die von Ihnen gewünschten Attribute verfügt. So erzeugen alle vordefinierten Stift-Werkzeuge nur dünne Linien. Wenn Sie nun aber dicke Linien zeichnen wollen, haben Sie zwei Möglichkeiten zur Erstellung des gewünschten Werkzeugs.

Sie können die Windows-Funktionen *CreatePen*, *CreateFont*, *CreateSolidBrush* oder *CreateDIBPatternBrush* aufrufen. Diese Funktionen haben Parameter, die das gewünschte Werkzeug beschreiben und Werkzeug-Handles zurückliefern, die dann in den *SelectObject*-Aufrufen eingesetzt werden können.

Die zweite Möglichkeit zum Erstellen eigener Werkzeuge besteht darin, daß Sie eine Beschreibung für die Attribute eines *virtuellen* Werkzeugs aufbauen. Ein virtuelles Werkzeug wird durch die Windows-Datenstrukturen *LOGPEN*, *LOGBRUSH*, *LOGFONT* und *LOGPALETTE* beschrieben. So verfügt zum Beispiel *LOGPEN* über Felder, in denen Stärke, Farbe und Art eines Stifts vermerkt sind.

Sobald Sie eine Datenstruktur für das virtuelle Werkzeug definiert haben, übergeben Sie diese als Parameter an *CreatePenIndirect*, *CreateBrushIndirect*, *CreateFontIndirect* oder *CreatePalette*. Diese Funktionen liefern Werkzeug- Handles zurück, die in *SelectObject*-Aufrufen verwendet werden können. Das folgende Beispiel setzt den Stift eines Bildschirmkontexts auf die Farbe Blau:

```
void SampleWindow::ChangePenToBlue()
{
  LOGPEN ALogPen;
  HPEN ThePen;

  ALogPen.lopnColor = RGB(0, 0, 255);
  ALogPen.lopnStyle = PS_SOLID;
  ALogPen.lopnWidth.X = 0;
  ALogPen.lopnWidth.Y = 0;
  ThePen = CreatePenIndirect(&ALogPen);
  SelectObject(TheDC, ThePen);
}
```

GDI-Zeichen-funktionen

In diesem Abschnitt werden die verschiedenen GDI-Funktionsaufrufe erläutert, die für das Zeichnen in Fenstern verfügbar sind.

Funktionen für das Zeichnen von Text

Die Funktionen für das Zeichnen von Text verwenden die aktuelle Schrift des angegebenen Bildschirmkontexts. Die Funktion *TextOut* zeichnet Text an einem angegebenen Punkt. *TextOut* richtet sich bei der Textausrichtung nach den Werten der aktuellen Textformat-Flags. Standardmäßig wird Text linksbündig ausgerichtet. Die aktuell gültige Ausrichtung kann mit der Funktion *GetTextAlign* ermittelt und mit der Funktion *SetTextAlign* festgelegt werden.

Die *TextOut*-Funktion ist die am häufigsten verwendete Funktion zum Zeichnen von Text. Die folgende *Paint*-Elementfunktion zeichnet, unter Verwendung der voreingestellten Textformat-Flags, ein linksbündig ausgerichtetes Zeichen-Array, dessen obere linke Ecke an der Position (10,15) des Client-Bereichs eines Fensters liegt.

```
void TMyWindow::Paint(HDC PaintDC, PAINTSTRUCT& PaintInfo)
{
  char MyTextString[20];

  strcpy(MyTextString, "Hello, World");
  TextOut(PaintDC, 10, 15, MyTextString, strlen(MyTextString));
}
```

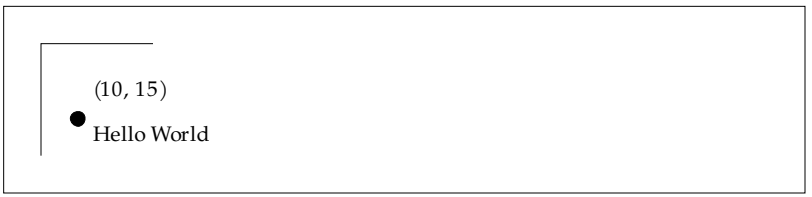


Abbildung 19.3 Das Ergebnis der Funktion TextOut

Funktionen für das Zeichnen von Linien

Die GDI-Funktionen für das Zeichnen von Linien verwenden den aktuellen Stift des angegebenen Bildschirmkontexts. Die überwiegende Mehrzahl aller Linien wird mit den Funktionen *MoveTo* und *LineTo* gezeichnet. Diese Funktionen beeinflussen das Bildschirmkontext-Attribut *aktuelle Position*. Die aktuelle Position entspricht beim Zeichnen mit Papier und Bleistift dem Punkt, an dem der Bleistift auf das Papier aufgesetzt wird.

MoveTo und LineTo

Die Funktion *MoveTo* verschiebt die aktuelle Position an den angegebenen Koordinatenpunkt. Die Funktion *LineTo* zeichnet eine Linie mit dem Stift von der aktuellen Position bis unmittelbar vor den angegebenen Koordinatenpunkt. Die aktuelle Position wird anschließend entsprechend aktualisiert.

Die folgende *Paint*-Elementfunktion zeichnet eine Linie von (100, 150) nach (10, 15):

```
void TMyWindow::Paint(HDC PaintDC, PAINTSTRUCT& PaintInfo) {
  MoveTo(PaintDC, 100, 150);
  LineTo(PaintDC, 10, 15);
}
```

Abbildung 19.4 Das Ergebnis der Funktion LineTo

Polyline

Die Funktion *Polyline* zeichnet mehrere Linien durch Verbinden von Punkten. Ähnliches würde erreicht, wenn zuerst die Funktion *MoveTo* und danach mehrmals die Funktion *LineTo* aufgerufen würde. *Polyline* führt die Operation

jedoch erheblich schneller aus und läßt die aktuelle Position unverändert. Die folgende *Paint*-Elementfunktion zeichnet einen rechten Winkel:

```
void TMyWindow::Paint(HDC PaintDC, PAINTSTRUCT& PaintInfo) {
  POINT Points[3];

  Points[0].x = 10;
  Points[0].y = 15;
  Points[1].x = 10;
  Points[1].y = 150;
  Points[2].x = 100;
  Points[2].y = 150;
  Polyline(PaintDC, Points, 3);
}
```

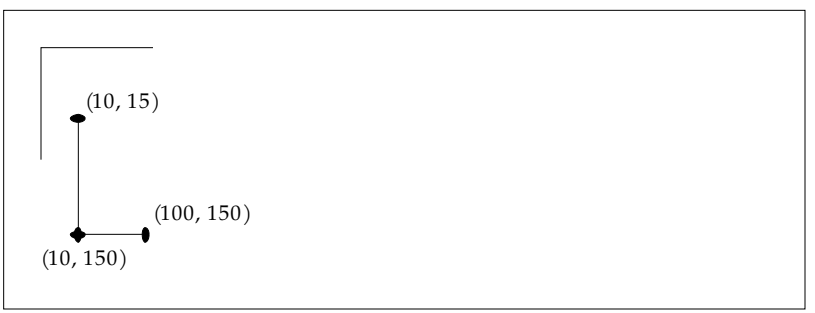


Abbildung 19.5 Das Ergebnis der Funktion Polyline

Arc

Die Funktion *Arc* zeichnet einen Bogen entlang des Umfangs der Ellipse, die von dem angegebenen Rechteck umgeben wird. Der Bogen beginnt am Schnittpunkt des Umfangs der Ellipse mit der Linie, die vom Mittelpunkt zum angegebenen Startpunkt führt. Der Bogen wird gegen den Uhrzeigersinn gezeichnet, bis er den Punkt erreicht, an dem die Ellipse sich mit der Linie schneidet, die vom Mittelpunkt der Ellipse zum angegebenen Endpunkt führt. *Arc* läßt die aktuelle Position unverändert.

Die folgende *Paint*-Elementfunktion zeichnet das obere Viertel eines Kreises. Dazu werden das umgebende Rechteck (10, 10) und (40, 40), der Startpunkt (0, 0) und der Endpunkt (50, 0) verwendet. Der Bogen wird dargestellt, obwohl die angegebenen Start- und Endpunkte nicht auf dem Bogen liegen.

```
void TMyWindow::Paint(HDC PaintDC, PAINTSTRUCT& PaintInfo) {
  Arc(PaintDC, 10, 10, 40, 40, 50, 0, 0, 0);
}
```

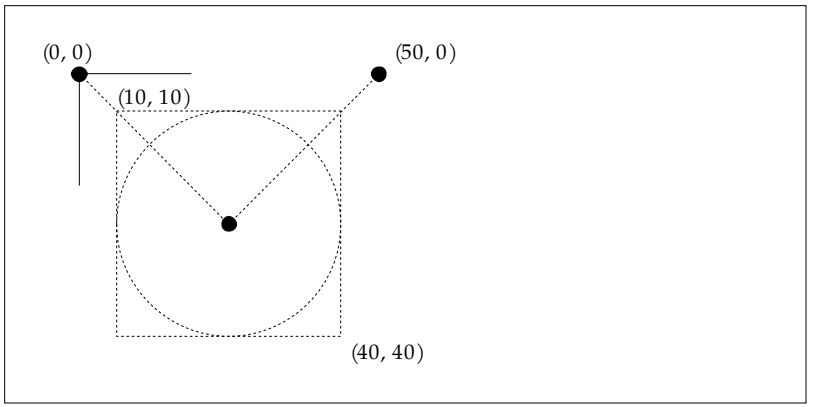


Abbildung 19.6 Das Ergebnis der Funktion Arc

Funktionen zum Zeichen von geometrischen Formen

Diese GDI-Funktionen verwenden den aktuellen Stift des angegebenen Bildschirmkontexts, um eine Umrahmung zu zeichnen, und den aktuellen Pinsel, um das Innere der Form auszufüllen. Die aktuelle Position ändert sich dabei nicht.

Rectangle

Die Funktion *Rectangle* zeichnet ein Rechteck mit der angegebenen oberen linken und unteren rechten Ecke.

Die folgende Anweisung in einer *Paint*-Elementfunktion zeichnet ein Rechteck von (10, 15) mit den Eckpunkten (100, 150):

```
Rectangle(PaintDC, 10, 15, 100, 150);
```

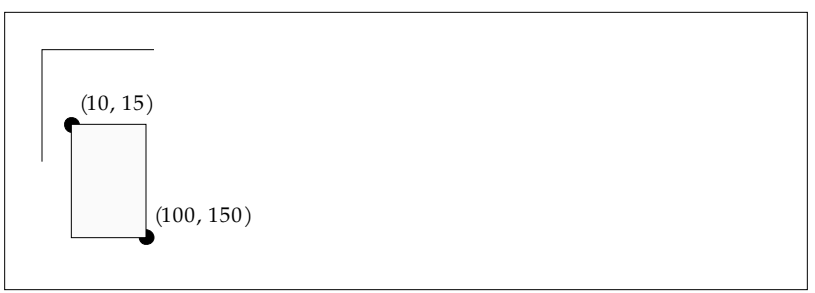


Abbildung 19.7 Das Ergebnis der Funktion Rectangle

RoundRect

Die Funktion *RoundRect* zeichnet ein Rechteck mit abgerundeten Ecken. Die abgerundeten Ecken sind als Viertel einer Ellipse definiert. Im folgenden Beispiel zeichnet die Anweisung in einer *Paint*-Elementfunktion ein Rechteck von (10,

15) zu (100, 150), wobei die Ecken die Viertel einer Ellipse mit einer Breite von 9 und einer Höhe von 11 sind.

```
RoundRect(PaintDC, 10, 15, 100, 150, 9, 11);
```

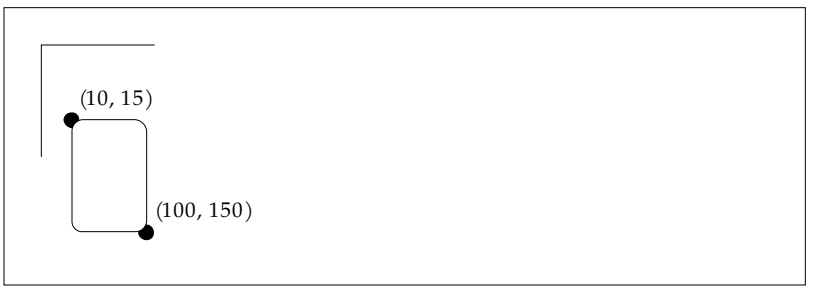


Abbildung 19.8 Das Ergebnis der Funktion RoundRect

Ellipse

Die Funktion *Ellipse* zeichnet eine Ellipse, die von dem umgebenden Rechteck definiert wird.

Das folgende Beispiel zeichnet eine Ellipse innerhalb des Rechtecks mit den Eckpunkten (10, 15) und (100, 150):

```
Ellipse(PaintDC, 10, 15, 100, 150);
```

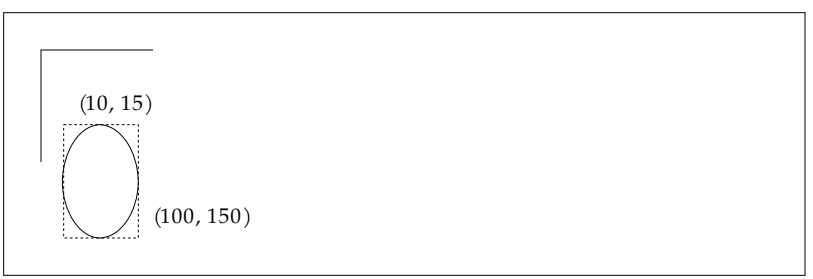


Abbildung 19.9 Das Ergebnis der Funktion Ellipse

Pie und Chord

Die Funktionen *Pie* und *Chord* zeichnen Ellipsen-Segmente. Sie zeichnen jedoch nicht nur den Rand der Ellipse (bzw. des Kreises), sondern auch einen Anteil vom Ellipseninneren.

Die Funktion *Pie* verbindet den Mittelpunkt der Ellipse mit den Endpunkten des Bogens.

Die folgende *Pie*-Funktion zeichnet das obere Viertel eines Kreises innerhalb des Rechtecks (10, 10), (40, 40).

```
Pie(PaintDC, 10, 10, 40, 40, 50, 0, 0, 0);
```

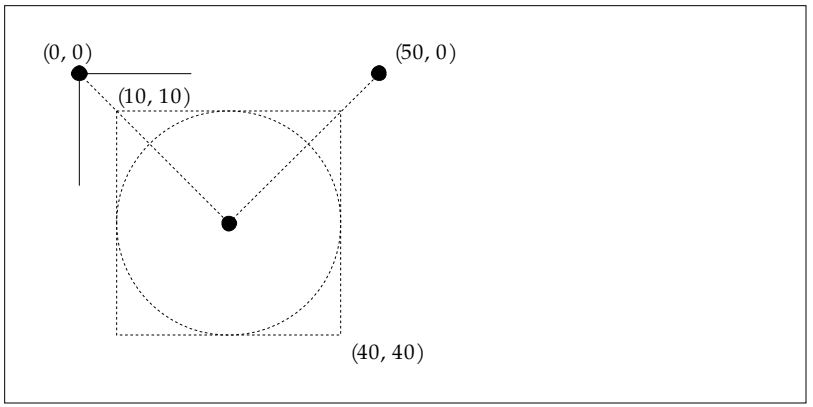


Abbildung 19.10 Das Ergebnis der Funktion Pie

Die Funktion *Chord* verbindet die beiden Eckpunkte des Kreisbogens:

```
Chord(PaintDC, 10, 10, 40, 40, 50, 0, 0, 0);
```

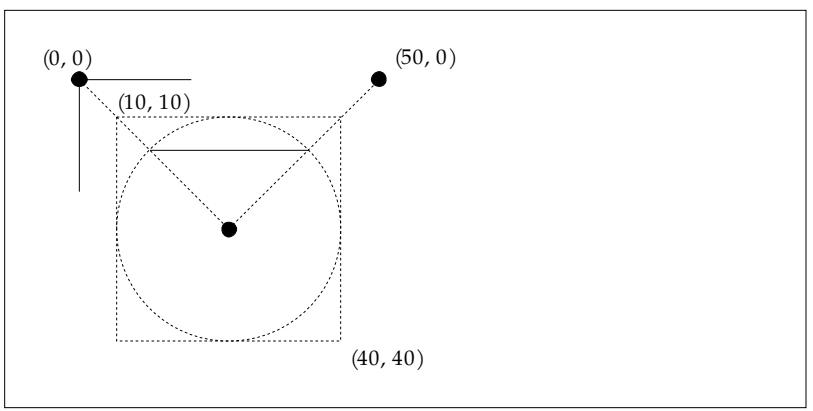


Abbildung 19.11 Das Ergebnis der Funktion Chord

Polygon

Die Funktion *Polygon* zeichnet zusammenhängende Liniensegmente, ähnlich wie die Funktion *Polyline*. Der Unterschied beruht darin, daß *Polygon* die Form schließt, indem ein Liniensegment vom angegebenen letzten zum angegebenen ersten Punkt gezogen wird. Schließlich wird das Polygon mit dem aktuellen Pinsel ausgefüllt, wobei der aktuelle Polygon-Füllmodus verwendet wird.

Die folgende *Paint*-Elementfunktion zeichnet ein rechtwinkeliges Dreieck.

```
void TMyWindow::Paint(HDC PaintDC, PAINTSTRUCT& PaintInfo)
{

  POINT Points[3];
```

```
  Points[0].x = 10;
  Points[0].y = 15;
  Points[1].x = 10;
  Points[1].y = 150;
  Points[2].x = 100;
  Points[2].y = 150;
  Polygon(PaintDC, Points, 3);
}
```

Abbildung 19.12 Das Ergebnis der Funktion Polygon

Der Einsatz von Paletten

Einige Ausgabegeräte können viele Farben anzeigen, aber nur einige davon zur gleichen Zeit. Die System- oder physikalische Palette enthält die Farben, die aktuell für die gleichzeitige Anzeige auf dem Ausgabegerät verfügbar sind. Windows gibt Ihrer Anwendung die teilweise Kontrolle über die Farben, die in die Systempalette des Ausgabegeräts übernommen werden. Wenn Ihre Anwendungen nur einfache Farben einsetzen, muß keine Palette direkt verwendet werden.

Die Änderung der Systempalette betrifft jedoch alle Ausgaben auf den Bildschirm und sogar andere Anwendungen. Das kann dazu führen, daß alle anderen Anwendungen mit falschen Farben angezeigt werden. Der Windows-Paletten-Manager löst dieses Problem, indem er als Vermittler zwischen den Anwendungen fungiert, die die Systempalette ändern wollen. Windows weist jeder Anwendung eine virtuelle Palette zu, welche die für die Anwendung erforderlichen Farben enthält. Der Paletten-Manager vergleicht die in der virtuellen Palette angeforderten Farben mit den in der Systempalette vorhandenen Farben. Ist die angeforderte Farbe nicht in der Systempalette, fügt der Manager diese in die Systempalette ein. Wenn die Systempalette keine zusätzlichen Farben mehr aufnehmen kann, werden diese Farben der ähnlichsten Farbe in der Systempalette zugeordnet.

Eine aktivierte Anwendung kann die Systempalette mit Farben aus ihrer virtuellen Palette ergänzen. Das hat unter Umständen die Folge, daß von anderen Anwendungen eingebrachte Farben »hinausgedrängt« werden. Gene-

rell reserviert Windows 20 permanente Farben in der Systempalette, die das Farbschema für alle Anwendungen und für Windows selbst aufrechterhalten.

Erstellen einer Farbpalette

Genauso wie Stifte und Pinsel (vgl. den Abschnitt »Zeichenwerkzeuge« weiter vorne in diesem Kapitel) sind auch virtuelle Paletten Zeichenwerkzeuge, die Sie mit der Funktion *CreatePalette* erzeugen können. Diese Funktion liefert ein Handle für eine virtuelle Palette zurück, das an *SelectPalette* zur Auswahl für einen Bildschirmkontext übergeben werden kann. An *CreatePalette* übergeben Sie einen Zeiger auf eine *LOGPALETTE*-Struktur. *LOGPALETTE* enthält Header-Informationen und ein Array mit Paletteneinträgen. Jeder Paletteneintrag ist eine *PALETTEENTRY*-Struktur, die aus drei Elementen für die Farbfestlegung (*peRed*, *peGreen* und *peBlue*) und aus einem Element für Flags (*peFlags*) besteht. Der folgende Code erzeugt eine voreingestellte Palette mit den 20 Farben, die immer in der Systempalette vorhanden sind.

```
GetStockObject (DEFAULT_PALETTE)
```

Wenn eine Anwendung ihre virtuelle Palette mit *SelectPalette* für den Bildschirmkontext ausgewählt hat, muß diese vor Gebrauch »realisiert« werden. Dies führt die Windows-Funktion *RealizePalette* aus:

```
ThePalette = CreatePalette(&ALogPalette);
SelectPalette(TheDC, ThePalette, 0);
RealizePalette(TheDC);
```

RealizePalette überträgt die Farben aus der virtuellen Palette der Anwendung in die Systempalette des Ausgabegeräts. Als erstes gleicht Windows die Farben ab, die sich bereits in der Systempalette befinden, und fügt dann die neuen Farben ein, solange genügend Platz dafür vorhanden ist. Schließlich werden die Farben, die weder in der Systempalette vorhanden sind noch darin eingefügt werden können, auf die ähnlichsten Farben in der Systempalette abgebildet.

Die Anwendung sollte ihre Palette realisieren, bevor mit dem Zeichnen begonnen wird – genauso wie die anderen Zeichenwerkzeuge vor dem Zeichnen ausgewählt werden müssen.

Zeichnen mit Paletten

Sobald die Anwendungspalette realisiert ist, kann das Zeichnen mit den Farben aus der Palette beginnen. Die Palettenfarben können direkt oder indirekt spezifiziert werden. Zur direkten Bestimmung einer Palettenfarbe verwenden Sie einen Paletten-Index des Typs *COLORREF*. *COLORREF* ist ein *DWORD*, dessen höherwertiges Byte auf 1 gesetzt ist und dessen beide niederwertigen Bytes den Index eines Eintrags in der virtuellen Palette enthalten. So bestimmt zum Beispiel 0x01000009 den neunten Eintrag in einer virtuellen Palette. Dieser

Wert kann überall da eingesetzt werden, wo ein *COLORREF*-Argument erwartet wird, zum Beispiel:

```
ALogPen.lopnColor = 0x01000009;
```

Wenn Ihr Ausgabegerät eine volle 24-Bit-Farbdarstellung ohne Verwendung einer Systempalette zuläßt, beschränkt Sie ein Paletten-Index unnötigerweise auf die Farben in Ihrer virtuellen Palette. Diese Einschränkung können Sie umgehen, wenn Sie die Palettenfarben indirekt mit einem palettenbezogenen *COLORREF*-Wert bestimmen. Ein palettenbezogener *COLORREF*-Wert ist derselbe Wert wie ein expliziter RGB-*COLORREF*-Wert. Sein höherwertiges Byte ist jedoch mit 2 besetzt. Die drei niederwertigen Bytes enthalten den RGB-Farbwert. So bestimmt zum Beispiel 0x020000FF einen palettenbezogenen *COLORREF*-Wert für reines Rot.

Wenn das Ausgabegerät eine Systempalette unterstützt, gleicht Windows die RGB-Information mit der ähnlichsten Farbe in der ausgewählten virtuellen Palette ab. Wird die Systempalette nicht unterstützt, dann wird *COLORREF* wie ein expliziter RGB-Wert verwendet.

Informationen über eine Palette

Durch einen Aufruf von *GetPaletteEntries* können Sie sich über den Inhalt einer Palette informieren. *GetPaletteEntries* füllt ein angegebenes Array von *PALETTEENTRY*-Strukturen mit den Paletteneinträgen. Im Aufruf von *GetPaletteEntries* werden außerdem der Index des ersten Eintrags und die Anzahl der zurückzuliefernden Einträge übergeben.

Ändern einer Palette

Die Einträge in einer virtuellen Palette können auf zwei Arten geändert werden. Die Funktion *SetPaletteEntries* hat dieselben Argumente wie *GetPaletteEntries* und ersetzt die angegebenen Einträge durch diejenigen, auf die das PALETTEENTRY-Argument zeigt.

Beachten Sie, daß sich Änderungen erst dann auf die Systempalette auswirken, wenn *RealizePalette* aufgerufen wurde, und erst dann sichtbar werden, wenn der Client-Bereich erneut gezeichnet wird. Die Funktion *AnimatePalette* hat dieselben Argumente wie *SetPaletteEntries*, wird aber dann verwendet, wenn eine Anwendung die Palette schnell ändern und die Änderungen sofort sichtbar machen soll. Wenn *AnimatePalette* aufgerufen wird, werden die Paletteneinträge, deren *peFlags*-Feld auf *PC_RESERVED* gesetzt ist, durch die entsprechenden neuen Einträge ersetzt und sofort mit der Systempalette abgeglichen. Andere Einträge werden dadurch nicht beeinflußt.

Nehmen wir folgendes Beispiel an: Sie wollen die ersten zehn Einträge einer Palette ermitteln, ihre Werte durch Heraufsetzen des Rot-Anteils und Herabsetzen des Blau- und Grün-Anteils ändern und die Änderungen sofort in Kraft

treten lassen (vorausgesetzt, daß einige der Paletteneinträge mit *PC_RESERVED* gesetzt sind):

```
GetObject(ThePalette, sizeof(NumEntries), &NumEntries);
if (NumEntries >= 10) {
  GetPaletteEntries(ThePalette, 0, 10, &PaletteEntries);
  for (int i = 0; i <= 9; ++i)
  {
    PaletteEntries[i].peRed = PaletteEntries[i].peRed + 40;
    PaletteEntries[i].peGreen = PaletteEntries[i].peGreen
    -40;
    PaletteEntries[i].peBlue = PaletteEntries[i].peBlue - 40;
  }
  AnimatePalette(ThePalette, 0, 10, &PaletteEntries);
}
```

Derselbe Effekt könnte auch so erzielt werden:

```
SetPaletteEntries(ThePalette, 0, 10, &PaletteEntries);
RealizePalette(ThePalette);
```

Danach muß der Bildschirm neu gezeichnet werden, um die Farbänderungen anzuzeigen.

Palettenbotschaften beantworten

Wenn ein Fenster eine *WM_PALETTECHANGED*-Botschaft erhält, ist das ein Zeichen dafür, daß das aktive Fenster die Systempalette durch Realisieren seiner virtuellen Palette geändert hat. Das empfangende Fenster kann auf drei Arten antworten:

- Es kann nichts unternehmen (kein Zeitverlust, kann aber zu unkorrekter Farbgebung führen).
- Es kann seine virtuelle Palette realisieren und sich dann neu zeichnen (langsamer, aber die Farben sind dann so korrekt wie möglich).
- Es kann seine Palette realisieren und dann mit der Funktion *UpdateColors* den Client-Bereich mit der Systempalette aktualisieren.

UpdateColors ist schneller als das Neuzeichnen des Client-Bereichs, wobei aber die Farbgenauigkeit etwas leidet.

Das *WParam*-Feld in einer *WM_PALETTECHANGED*-Botschaft enthält das Handle für das Fenster, das seine Palette realisiert. Wenn Sie als Antwort die Palette realisieren, müssen Sie sicherstellen, daß dieses Handle nicht das eigene Fenster-Handle ist. Sie würden ansonsten eine Endlosschleife erzeugen.

Das Programm PALTEST.CPP, das im Verzeichnis EXAMPLES enthalten ist, erstellt und realisiert eine virtuelle Palette mit acht Farben. Klicken Sie mit dem linken Mausschalter, wird ein Quadrat in jeder Farbe gezeichnet, klicken Sie mit dem rechten Mausschalter, werden die Farben in der Palette rotiert. Es wird ein

COLORREF-Palettenindex verwendet. Wird daher die virtuelle Palette geändert, ändern sich die Farben der Quadrate, wenn sie neu gezeichnet werden.

Das Makro *PALETTEINDEX* wird für die Erstellung der *COLORREF*-Palettenindizes verwendet.

20. Details über Ressourcen

Windows-Programme sind aufgrund ihrer standardisierten Oberfläche sehr einfach zu bedienen. So verwenden beispielsweise die meisten Windows-Anwendungen Menüs für die Befehlsauswahl und Zeigersymbole, mit denen der Mauszeiger die verschiedensten Werkzeuge (Pfeile, Pinsel, usw.) darstellen kann.

Menüs und Mauszeiger sind nur zwei Beispiele für die Ressourcen einer Windows-Anwendung. Ressourcen sind Daten, die in der .EXE-Datei der Anwendung gespeichert sind, sich aber nicht im normalen Datensegment des Programms befinden. Sie werden außerhalb des Quelltextes definiert und erst zum compilierten Code für die zu erzeugende .EXE-Datei hinzugebunden. Die folgenden Ressourcen werden Sie am häufigsten erstellen und verwenden:

- Menüs
- Dialogfenster
- Symbole
- Mauszeiger
- Tastenkürzel
- Bitmaps
- Strings

Beim Laden eines Programms in den Arbeitsspeicher des Computers beläßt Windows die Ressourcen normalerweise auf der Festplatte. Benötigte Ressourcen werden bei Bedarf erst während der Programmausführung einzeln geladen. Wird die Ressource nicht mehr benötigt, entfernt sie Windows aus dem Speicher (dies gilt für alle Ressourcen mit Ausnahme der Bitmaps).

Soll eine Ressource gleichzeitig mit dem Programm geladen werden oder soll Windows sie nicht aus dem Arbeitsspeicher entfernen, müssen ihre Attribute entsprechend geändert werden.

Ressourcen erstellen

Ressourcen können mit Hilfe eines Ressourceneditors oder eines Ressourcencompilers erstellt werden. Borland C++ unterstützt beide Methoden, Sie können

also wählen, nach welcher Sie vorgehen wollen. In den meisten Fällen ist es einfacher, die Ressourcen mit Hilfe eines Editors auf dem Bildschirm zu erstellen. Die Verwendung eines Ressourcencompilers bietet sich dagegen an, wenn Ressourcen-Script-Dateien aus Büchern oder Zeitschriften compiliert werden sollen.

Gleichgültig, für welche Methode Sie sich entscheiden, wird für eine Anwendung normalerweise eine Ressourcendatei (.RES) erstellt. Diese Datei enthält Informationen in binärer Form für alle Menüs, Dialogfenster, Bitmaps und andere in der Anwendung eingesetzten Ressourcen.

Die binäre Ressourcendatei (.RES) wird während der Compilierung zur ausführbaren Datei (.EXE) hinzugefügt. Zum Laden der Ressourcen in den Speicher müssen entsprechende Befehle in den Quelltext aufgenommen werden. Um möglichst wenig Speicher zu belegen, wird jede Ressource separat geladen, sobald sie benötigt wird.

Das Laden von Ressourcen wird im vorliegenden Kapitel ausführlich beschrieben.

Ressourcen zu einer .EXE-Datei hinzufügen

Nachdem die Ressourcen in Form einer .RES-Datei im binären Format vorliegen, müssen sie zur .EXE-Datei des Programms hinzugefügt werden. Dadurch entsteht eine Datei, die sowohl den compilierten Quelltext als auch die Ressourcen der Anwendung enthält.

Es gibt zwei Möglichkeiten, Ressourcen zur .EXE-Datei hinzuzufügen:

1. Die Ressourcen werden mit dem Ressourceneditor von einer .RES-Datei in die bereits compilierte .EXE-Datei des Programms kopiert.

2. Das Kopieren der Ressourcen von einer .RES-Datei in die bereits compilierte .EXE-Datei des Programms erfolgt mit dem Ressourcencompiler (er wird im folgenden Abschnitt ausführlich beschrieben).

Der Ressourcen-compiler

Der vorliegende Abschnitt enthält einen kurzen Überblick über die Erstellung von Ressourcendateien mit dem Ressourcencompiler von Microsoft.

Der Ressourcencompiler ist eine DOS-Anwendung, die aus einer Ressourcen-Script-Datei eine binäre Ressourcendatei erzeugt. Es besteht auch die Möglichkeit, die Ressourcen mit dem Ressourcencompiler direkt zur .EXE-Datei hinzuzufügen. Der Start des Ressourcencompilers kann unter DOS oder in einem DOS-Fenster von Windows erfolgen. Das Format für den Start lautet:

```
RC [Optionen]  <.RC-Eingabedatei>  [.EXE-Ausgabedatei]
```

Die Einträge in eckigen Klammern sind optional. Sehen wir uns dazu ein Beispiel an:

```
RC SAMPPROG.RC
```

erzeugt aus der der Eingabe-Script-Datei SAMPPROG.RC eine SAMPPROG.RES-Datei und fügt diese direkt zu SAMPPROG.EXE hinzu. Die vorhandenen Ressourcen werden dadurch ersetzt. In jedem Fall muß das C++-Programm SAMPPROG.CPP Befehle zum Laden der Ressourcen in den Speicher enthalten.

Die folgende Tabelle listet die zulässigen Befehlszeilenoptionen für den Ressourcencompiler auf:

Tabelle 20.1 Befehlszeilenoptionen für den Ressourcencompiler

20 **Der Ressourcencompiler** 289

-r	Erzeugt eine .RES-Datei.
-l	Erzeugt eine Anwendung, die EMS nach LIM 3.2 verwendet.
-e	Erzeugt einen Treiber, der den EMS-Speicher nutzt.
-m	Setzt ein Flag für mehrere Instanzen.
-p	Erzeugt eine private Bibliothek.
-t	Erzeugt eine Anwendung, die nur im Protected Mode läuft.
-v	Protokoll (Meldungen über den Arbeitsfortschritt).
-d	Definiert ein Symbol.
-fo	Benennt eine .RES-Datei um.
-fe	Benennt eine .EXE-Datei um.
-i	Fügt einen Pfad für INCLUDE-Dateien hinzu.
-x	Ignoriert die Environment-Variable INCLUDE.
-k	Hält die Segmente in der .DEF-Dateireihenfolge bei (die Segmente werden nicht für schnelles Laden sortiert).

Ressourcen-Script-Dateien

Bei einer .RC-Script-Datei handelt es sich um eine Quelltextdefinition der Ressourcen, die in ein binäres Format compiliert werden müssen. Die Festlegung von Ressourcen im Format des Ressourcencompilers entspricht in etwa dem Schreiben eines Programms. Wir wollen an dieser Stelle nicht die Ressourcen-Script-Sprache im Detail präsentieren, sondern anhand einiger Beispiele zeigen, wie .RC-Dateien compiliert und editiert werden können. Vergessen Sie nicht, daß die Erstellung komplexer Ressourcen in den meisten Fällen mit einem Ressourceneditor leichter zu bewerkstelligen ist. Auch Script-Dateien können direkt mit dem Ressourceneditor erzeugt werden.

Das folgende Beispiel zeigt eine typische Ressourcen-Script-Datei mit dem Namen SAMPPROG.RC.

```
; SAMPPROG.RC --Beispiel-Ressourcen-Script-Datei.
;     In dieser Script-Datei wird die Verwendung von Symbol-,
;     String-, Menü und Dialog-Ressourcen demonstriert.
```

```
;     Sie muß mit dem Ressourcencompiler compiliert und
;     anschließend mit der entsprechenden Direktive dem
;     C++-Programm hinzugefügt werden.

; Header-Dateien für Konstantendefinitionen einbinden:
#include <windows.h>
#include <rcdefs.h>
#include <sampprog.h>

; Symbole sind in separaten Dateien definiert.
SampleIcon  ICON  sample.ico

; Strings können numerisch oder symbolisch definiert werden.
STRINGTABLE
BEGIN
  IDS_NOPROGRAM, "Program unavailable"
  IDS_INVALID, "Invalid entry"
  IDS_FATAL, "Fatal error!"
;  .
;  . weitere Strings sind hier definiert
;  .
END

; Menüs definieren Befehle für die Anwender.
SampleMenu MENU
BEGIN
  POPUP "&File"
    BEGIN
      MENUITEM "&New..\tCtrl+N",       CM_FILENEW
      MENUITEM "&Open..\tCtrl+O",      CM_FILEOPEN
      MENUITEM "&Save..\tCtrl+S",      CM_FILESAVE
      MENUITEM SEPARATOR
      MENUITEM "&Print \tCtrl+P",      CM_FILEPRINT
    END
  POPUP "&Edit"
    BEGIN
    MENUITEM "&Undo \tAlt+BkSp",       CM_EDITUNDO
    MENUITEM SEPARATOR
    MENUITEM "&Cut \tShift+Del",        CM_EDITCUT
    MENUITEM "&Copy \tIns",            CM_EDITCOPY
    MENUITEM "&Paste \tShift+Ins",     CM_EDITPASTE
    MENUITEM "&Delete \tDel",          CM_EDITDELETE
    MENUITEM "&Clear All \tCtrl+Del",  CM_EDITCLEAR
    END
  POPUP "&View"
    BEGIN
      MENUITEM "Summary \tF2",         CM_VIEWSUMMARY
      MENUITEM "Graph \tF3",           CM_VIEWGRAPH
    END
END
```

```
; Tastenkürzel für Menüs und Befehle.
SampleAccelerators ACCELERATORS
BEGIN
  "^n", CM_FILENEW
  "^o", CM_FILEOPEN
  "^s", CM_FILESAVE
  "^p", CM_FILEPRINT
  VK_DELETE, CM_EDITCUT, VIRTKEY, SHIFT
  VK_INSERT, CM_EDITCOPY, VIRTKEY
  VK_INSERT, CM_EDITPASTE, VIRTKEY, SHIFT
  VK_DELETE, CM_EDITDELETE, VIRTKEY
  VK_DELETE, CM_EDITCLEAR, VIRTKEY, CONTROL
  VK_BACK, CM_EDITUNDO, VIRTKEY, ALT
  VK_F2, CM_VIEWSUMMARY, VIRTKEY
  VK_F3, CM_VIEWGRAPH, VIRTKEY
END

; Dialogfenster blenden Informationen ein.
AboutBox DIALOG 20, 20, 160, 80
CAPTION "About SAMPPROG"
STYLE DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
FONT 8, "Helv"
BEGIN
  CTEXT "Sample Program"              -1,   0,  10,  160,  10
  CTEXT "Written in ObjectWindows"    -1,   0,  26,  160,  10
  CTEXT "for Windows 3.0"             -1,   0,  36,  160,  10
  ICON "SAMPLE_ICON"                  -1,   8,   8,    0,   0
  DEFPUSHBUTTON "OK",               IDOK,  60,  56,   40,  14
END

FileOpen DIALOG 20, 20, 204, 124
CAPTION "File Open"
STYLE DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
FONT 8, "Helv"
BEGIN
  LTEXT    "File&name:",        -1,   6,   8,  36,  10
  EDITTEXT                     100,  42,   6,  98,  12, ES_AUTOH-
SCROLL
  LTEXT    "Directory:",        -1,   6,  20,  36,  10
  LTEXT    ""                  101,  42,  20,  98,  10
  LTEXT    "&Files:",           -1,   6,  32,  64,  10
  LISTBOX  102, 6, 44, 64, 82, WS_TABSTOP | WS_VSCROLL | LBS_SORT
  LTEXT    "&Directories:",     -1,  76,  32,  64,  10
  LISTBOX  103, 76, 44, 64, 82, WS_TABSTOP | WS_VSCROLL | LBS_SORT
  DEFPUSHBUTTON "OK",         IDOK, 146,  6,  50,  14
  PUSHBUTTON    "Cancel", IDCANCEL, 146, 24,  50,  14
END
```

Die Header-Datei sampprog.h enthält Konstantendefinitionen im C-Stil. Eine typische Header-Datei sieht folgendermaßen aus:

```
/* SAMPPROG.H    --Header-Datei mit Konstanten für SAMPPROG.RC /*
#define IDS_NOPROGRM       601
#define IDS_INVALID        602
#define IDS_FATAL          603
#define CM_FILENEW         701
#define CM_FILESAVE        702
/*
  .
  . hier folgen weitere Konstanten
  .
*/
```

Hinweise für die Benutzung des Ressourcencompilers

Bei der Arbeit mit der Ressourcen-Script-Sprache ist folgendes zu beachten:

1. Kommentare beginnen mit einem Strichpunkt (;) und enden am Zeilenende oder werden durch /* und */ eingeschlossen.

2. Der Ressourcencompiler unterscheidet bei Schlüsselwörtern wie BEGIN und END *nicht* zwischen Groß- und Kleinschreibung.

3. Bei symbolischen Namen wie *CMFILENEW* und *AboutBox* wird zwischen Groß- und Kleinschreibung unterschieden.

4. Der Ressourcencompiler überspringt überzählige Whitespace-Zeichen.

5. Zur Aufnahme von Header-Dateien mit Konstanten muß die #include-Direktive verwendet werden. Diese Konstanten werden mit einem #define-Befehl definiert.

6. Zur Aufnahme anderer Ressourcen-Script-Dateien (*.RC oder *.DLG) kann die #rcinclude-Direktive verwendet werden. Auf diese Weise kann zum Beispiel das Datei-Dialogfenster von ObjectWindows einbezogen werden.

7. Jede Menüeintrags-ID bzw. String-ID muß eindeutig sein.

8. Menüs können beliebig verschachtelt werden.

9. In Menüpunkten und Steuerelementen von Dialogen kann ein Zeichen durch Voranstellen des et-Zeichens (&) unterstrichen werden. Der betreffende Menüpunkt bzw. das Steuerelement kann dann durch gleichzeitiges Drücken der Taste Alt und des unterstrichenen Zeichens aufgerufen werden.

10. In Menüpunkten kann zur Ausrichtung von Text mit `\t` ein Tabulatorzeichen erzeugt werden. Auf diese Weise können zum Beispiel Tastaturbefehl-Äquivalente ausgerichtet werden.

11. Tastenkürzel können in einer Tastenkürzeltabelle definiert werden.

12. Steuerelemente, auf die das Programm nicht zugreift (zum Beispiel statischer Text), haben normalerweise die ID -1.

13. Die Position von Dialogfenstern und Steuerelementen wird in der Reihenfolge Links, Oben, Breite, Höhe angegeben.

14. Menüpunkte, Dialogfenster und Steuerelemente wie Bildlaufleisten, Editierfelder und Schalter können mit Windows-Stil-Konstanten modifiziert werden. Diese Konstanten werden mit dem bitweisen ODER-Operator, einem senkrechten Strich (|) hinzugefügt.

Um beispielsweise Steuerelementen die Stile *WS_CHILD*, *WS_TABSTOP* und *WS_BORDER* zuzuordnen, geben Sie WS_CHILD | WS_TABSTOP | WS_BORDER an.

15. Meist ist es einfacher, einen Ressourceneditor zu verwenden, anstatt komplexe Ressourcendefinitionen mit Script-Dateien zu schreiben.

16. Im Programm müssen Befehle enthalten sein, die das Laden und die Verwendung der Ressourcen veranlassen.

Ressourcen in einer Anwendung laden

Nachdem eine Ressource zur ausführbaren Datei hinzugefügt wurde, muß sie von der Anwendung explizit geladen werden. Die Vorgehensweise beim Laden hängt von der Art der jeweiligen Ressource ab.

Menüs laden

Das Menü eines Fensters ist ein Erstellungsattribut des Fensters, d.h. es muß vor dem entsprechenden Fensterelement mit einer *Create*-Elementfunktion erzeugt werden. Ein Menü muß daher im Fensterklassen-Konstruktor oder kurz nach der Initialisierung angegeben werden. Das Laden einer Menü-Ressource erfolgt bei der Erstellung eines neuen Fensterobjekts durch einen Aufruf der (von *TWindow* geerbten) Elementfunktion *AssignMenu*. Dabei muß der ID-String für das Menü angegeben werden:

```
SampleMainWindow::SampleMainWindow(PTWindowsObject AParent,
                    LPSTR ATitle): TWindow(AParent, ATitle)
{
  AssignMenu(100);
  ...
}
```

AssignMenu lädt die Menü-Ressource mit einer ID von 100 in das neue Fensterobjekt. Eine Ressource muß nicht unbedingt eine numerische ID haben, sie kann mit einem symbolischen Namen (einem String) bezeichnet werden, zum Beispiel 'SampleMenu'. In diesem Fall müß te der obige Quelltext folgendermaß en geschrieben werden:

```
SampleMainWindow::SampleMainWindow(PTWindowsObject AParent,
                                   LPSTR ATitle):
                                   TWindow(AParent, ATitle)
{
  AssignMenu("SampleMenu");
  ...
}
```

Das Erstellen von Fensterobjekten wird in Kapitel 10 »Fensterobjekte« ausführlich beschrieben. Zur Bearbeitung einer Menüauswahl wird eine Elementfunktion für das Fenster definiert, zu dem das Menü gehört. Dazu wird der Kopf der Elementfunktion mit dem *CM_FIRST*-Bezeichner erweitert:

```
virtual void HandleMenu101(RTMessage Msg) = [CM_FIRST + 101];
```

101 ist die ID der Menüauswahl, nicht diejenige der Menü-Ressource. Jede Menüauswahl hat eine eindeutige, aus einem Integer bestehende ID. In dieser Elementfunktion müssen Aufgaben definiert werden, die auf die Menüauswahl antworten. Normalerweise werden Sie für Ihre Menübefehle symbolische Konstanten definieren.

Tastenkürzeltabellen laden

Tastenkürzel dienen dem schnellen Aufruf von Befehlen. Sie werden meist als als Abkürzung für Menübefehle definiert. Die Entf -Taste wird zum Beispiel häufig als Alternative zur Auswahl des Löschen-Befehls in einem Bearbeiten-Menü verwendet. Tastenkürzel können aber auch Befehle implementieren, die nicht in einem Menü enthalten sind.

Tastenkürzel-Ressourcen werden in einer Tastenkürzeltabelle gespeichert. Das Laden einer Tastenkürzeltabelle erfolgt durch einen Aufruf der Windows-Funktion *LoadAccelerators*, die ein Handle auf die Tabelle zurückliefert. Im Gegensatz zu einer Menü-Ressource, die mit einem bestimmten Fenster verknüpft ist, gehört die Tastenkürzel-Ressource zur gesamten Anwendung. Jede Anwendung kann nur eine Tastenkürzeltabelle haben.

Anwendungsobjekte reservieren zum Speichern des Handle der Tastenkürzel-Ressource ein Datenelement namens *HAccTable*. Normalerweise erfolgt das Laden der Tastenkürzel-Ressource in der *InitInstance*-Elementfunktion des Anwendungsobjekts:

```
void SampleApplication::InitInstance()
{
  TApplication::InitInstance();
  HAccTable = LoadAccelerators(GetApplication()->HInstance,
                               "SampleAccelerators");
}
```

Meist werden Sie die Tastenkürzel für Menüauswahlen definieren. Zum Beispiel wird Umschalt Einfg häufig als Kürzel für den Menüpunkt *Einfügen*

verwendet. Die Eingabe eines Tastenkürzels erzeugt eine Befehlsbotschaft, die der bei der Auswahl eines Menüpunkts entspricht. Um die Botschaftsantwortfunktion für eine Menüauswahl mit einem Tastenkürzel zu verknüpfen, muß der Wert des Tastenkürzels in der Ressource mit der ID der Menüauswahl übereinstimmen.

Dialogfenster laden

Dialogfenster haben als einziger Ressourcentyp korrespondierende Klassen in ObjectWindows. *TDialog* und die von *TDialog* abgeleiteten Klassen definieren Schnittstellenobjekte, die Dialogfenster-Ressourcen verwenden. Jedes Dialogfensterobjekt ist normalerweise mit einer Dialogfenster-Ressource verknüpft, die seine Größe, seine Position und die Anordnung seiner Steuerelemente (beispielsweise Schalter und Listen) festlegt.

Die Ressource für ein Dialogfensterobjekt wird bei der Erstellung des Dialogfensters angegeben. Dialogfenster-Ressourcen können – ebenso wie Menü- oder Tastenkürzel-Ressourcen – einen symbolischen Namen oder eine Integer-ID haben:

```
ADLg = new TSampleDialog(this, "AboutBox");
```

oder

```
ADLg = new TSampleDialog(this, 120);
```

Die Erstellung von Dialogfenstern wird in Kapitel 5 »Einen Dialog erstellen« ausführlich erläutert.

Mauszeiger und Symbole laden

Zu jeder Fensterklasse gehören spezielle Attribute, die sogenannten *Registrierungsattribute*. Dazu gehören zum Beispiel die Mauszeiger und Symbole des Fensters. Zum Setzen dieser Attribute definieren Sie eine Elementfunktion mit dem Namen *GetWindowClass* (und eine namens *GetClassName*).

Angenommen, Sie haben ein Zeigersymbol zur Auswahl von Einträgen in einer Liste erstellt. Das Zeigersymbol hat die Form eines Zeigefingers und ist in einer Mauszeiger-Ressource mit dem Namen »Finger« gespeichert. Außerdem haben Sie ein Symbol namens »SampleIcon« erzeugt, das ein Gesicht darstellt.

Zum Laden dieser Ressourcen schreiben Sie die folgende *GetWindowClass*-Elementfunktion:

```
void SampleWindow::GetWindowClass(WNDCLASS& AWndClass)
{
  TWindow::GetWindowClass(AWndClass);
  AWndClass.hCursor = LoadCursor(GetApplication()->HInstance,
                                 "Finger");
```

```
  AWndClass.hIcon = LoadIcon(GetApplication()->HInstance,
                             "SampleIcon");
}
```

Zwischen Mauszeigern und Symbolen existiert folgender Unterschied: Während ein Mauszeiger nur für ein bestimmtes Fenster definiert wird, repräsentiert ein Symbol die gesamte Anwendung. Das Symbol wird deshalb nur im Hauptfensterobjekt eingerichtet. Von dieser Regel gibt es allerdings zwei Ausnahmen:

- Folgt die Anwendung dem Standard für die Multidokumentschnittstelle (MDI), hat jedes untergeordnete MDI-Fenster sein eigenes Symbol.
- Jedes Pop-up-Fenster der Anwendung hat sein eigenes Symbol.

Wenn Sie die vordefinierten Mauszeiger und Symbole von Windows verwenden wollen, müssen Sie anstelle von *HInstance* eine Null übergeben und einen *IDC_*-Wert (z.B. *IDC_IBEAM*) für einen Mauszeiger bzw. einen *IDI_*-Wert (z.B. *IDI_HAND*) für ein Symbol verwenden. Dazu ein Beispiel:

```
void SampleWindow::GetWindowClass(WNDCLASS& AWndClass)
{
  TWindow::GetWindowClass(AWndClass);
  AWndClass.hCursor = LoadCursor(NULL, IDC_IBEAM);
  AWndClass.hIcon = LoadIcon(NULL, IDI_HAND);
}
```

Auf Registrierungsattribute für Fenster wird in Kapitel 10 »Fensterobjekte« ausführlich eingegangen.

String-Ressourcen laden

Die Strings einer Anwendung werden als Ressourcen definiert, um die Anpassung der Anwendung an bestimmte Gegebenheiten zu vereinfachen oder ihre Übersetzung in andere Sprachen zu erleichtern. Wenn die Strings im Quelltext enthalten sind, muß für jede Änderung oder Übersetzung auf den Quelltext zugegriffen werden. Sind sie dagegen als Ressourcen definiert, werden sie in einer Stringtabelle innerhalb der .EXE-Datei einer Anwendung gespeichert. Die Strings können dann mit dem String-Editor des Ressourcencompilers übersetzt und geändert werden, ohne daß auf den Quelltext zugegriffen werden muß. Zu jeder .EXE-Datei kann jeweils nur eine Stringtabelle existieren.

Mit der Funktion *LoadString* kann ein String aus der Stringtabelle in einen Puffer des Datensegments der Anwendung geladen werden. Die Syntax für *LoadString* lautet:

```
LoadString(GetApplication -> HInstance, StringID, TextItem,
           MaxChars);
```

- Der Parameter *StringID* ist die ID des Strings (ein Integerwert) in der Stringtabelle. Anstelle der Nummer kann auch eine Konstante verwendet werden.

- Der Parameter *TextItem* ist ein *LPSTR*-Zeiger auf den Puffer.

- Der Parameter *MaxChars* gibt die maximale Anzahl der Zeichen an, die in den Puffer kopiert werden können. Die maximale Größe einer String-Ressource liegt bei 255 Zeichen. Wenn Sie sicherstellen wollen, daß der gesamte String eingetragen wird, geben Sie 256 für die Größe des Zeichenpuffers an.

LoadString liefert die Anzahl der Zeichen zurück, die in den Puffer kopiert wurden. Existiert die Ressource nicht, liefert *LoadString* Null zurück.

Mit Hilfe einer String-Ressource können Sie Text in einem Meldungsfenster anzeigen. Angenommen, es soll die Fehlermeldung »Programunavailable« angezeigt werden. Dazu wird der entsprechende String in einer Stringtabelle definiert und seine ID als Konstante *IDS_NOPRGRM* festgelegt.

Damit diese Meldung als String-Ressource verwendet werden kann, wird folgende Prozedur geschrieben:

```
virtual void TestDialog::RunErrorBox(int ErrorNumber) {
  char TextItem[255];

  LoadString(GetApplication()->HInstance, IDS_NOPRGRM,
             TextItem, 20);
  MessageBox(HWindow, TextItem, "Error", MB_OK |
             MB_ICONEXCLAMATION);
}
```

Diese Prozedur lädt einen einzelnen String in ein Fehlermeldungsfenster. Soll eine Stringliste in eine Liste geladen werden, muß jeder String einzeln mit *LoadString* geladen und dann mit *AddString* der Liste hinzugefügt werden.

Mit Hilfe von String-Ressourcen können auch Menüauswahlen innerhalb Ihres Quelltextes an vorhandene Menüs angehängt werden. Dazu wird zunächst der String mit *LoadString* geladen und anschließend in Aufrufen der Windows-Funktionen *CreateMenu* und *AppendMenu* als Parameter übergeben:

```
SampleWindow::SampleWindow(PTWindowsObject AParent, LPSTR
                           ATitle) : TWindow(AParent, ATitle)
{
  char TextItem[255];

  Assign(100);
  LoadString(GetApplication()->HInstance, 301, TextItem, 20);
  AppendMenu(Attr.Menu, MF_STRING | MF_ENABLED, 501, TextItem);
}
```

Bitmaps laden

Bitmap-Ressourcen werden mit der Windows-Funktion *LoadBitmap* in den Speicher geladen. Die Funktion liefert ein Handle für das Bitmap zurück.

Zum Beispiel lädt

```
HMyBit = LoadBitmap(GetApplication()->HInstance,
                    MAKEINTRESOURCE(501));
```

die Bitmap-Ressource mit der ID 501 und speichert das zurückgelieferte Bitmap-Handle in der Variable *HMyBit*.

Ist ein Bitmap einmal geladen, bleibt es im Speicher, bis es explizit gelöscht wird, auch wenn der Anwender das Programm beendet.

Windows verfügt über eine Anzahl vordefinierter Bitmaps, die Bestandteil der grafischen Schnittstelle von Windows sind. Diese Bitmaps, zum Beispiel *OBM_DNARROW*, *OBM_CLOSE* oder *OBM_ZOOM*, können von Ihrer Anwendung geladen werden. Zum Laden vordefinierter Bitmaps können Sie – wie bei vordefinierten Symbolen und Mauszeigern – im *LoadBitmap*-Aufruf Null für den Parameter *GetApplication()->HInstance* einsetzen:

```
HMyBit = LoadBitmap(NULL, OBM_CLOSE);
```

Nachdem das Bitmap geladen wurde, kann es in der Anwendung auf verschiedene Weise eingesetzt werden:

- Zum Zeichnen eines Bildes auf dem Bildschirm. Beispielsweise kann zum Zeichnen eines Logos ein Bitmap in das *About*-Fenster einer Anwendung geladen werden.
- Zum Erstellen eines Pinsels. Mit dem Pinsel kann dann ein Bereich des Bildschirms gefüllt oder der Hintergrund eines Fensters (zum Beispiel ein Streifenmuster) erzeugt werden.
- Zum Anzeigen von Bildern anstelle von Text in Menüs oder von Einträgen in Listen. Beispielsweise kann anstelle des Worts »Pfeil« für eine Menüauswahl das Bild eines Pfeiles angezeigt werden.

In Kapitel 19 »Die graphische Geräteschnittstelle« wird die Verwendung von Bitmaps ausführlich erläutert.

Wenn das Bitmap nicht mehr benötigt wird, sollte es aus dem Speicher gelöscht werden, da der belegte Speicherplatz sonst nicht von anderen Anwendungen genutzt werden kann. Löschen Sie das Bitmap spätestens vor der Beendigung der Anwendung. Zum Löschen wird die Windows-Funktion *DeleteObject* verwendet:

```
if DeleteObject(HMyBit)) ...        // erfolgreich
```

Nach dem Löschen des Bitmaps ist das Handle ungültig und kann nicht mehr verwendet werden.

Einen Pinsel mit Hilfe eines Bitmaps erstellen

Zum Füllen einer Fläche auf dem Bildschirm können Sie mit Hilfe eines Bitmaps einen Pinsel erstellen. Die Fläche kann einfarbig sein oder mit einem Muster gefüllt werden.

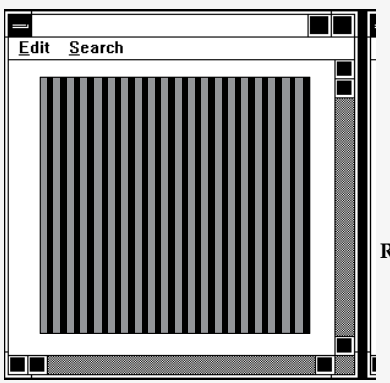
Ein Bitmap, das als Pinsel benutzt werden soll, muß mindestens eine Fläche

von 8 mal 8 Pixeln abdecken. Ist das Bitmap größer, wird nur die 8 mal 8 Pixel große Fläche in seiner linken oberen Ecke als Pinsel verwendet.

Angenommen, Sie wollen eine Fläche mit einem Streifenmuster füllen, wie in Abbildung 20.1 gezeigt. Dazu erstellen Sie mit einem Ressourceneditor das zweistreifige Muster (vgl. Abbildung 20.2). Es muß mindestens 8 mal 8 Pixel groß sein.

Anschließend können Sie das Bitmap in Ihrem Quelltext verwenden und damit einen Pinsel erstellen. Zum Füllen der gesamten Fläche (wie in Abbildung 20.1 gezeigt) kopiert Windows den Pinsel mehrmals.

20 **Ressourcen in einer Anwendung laden** *299*



Das eigentliche Bitmap hat nur eine Größe von 8 mal 8 Pixeln, mit Hilfe des Pinsels kann aber der gesamte Bildschirm gefüllt werden.

Das folgende Programm legt das Bitmap-Muster für den Pinsel fest:

```
void SampleWindow::MakeBrush() {
  LOGBRUSH  MyLogBrush;

  HMyBit = LoadBitmap(GetApplication()->HInstance,
                      MAKEINTRESOURCE(502));
  MyLogBrush.lbStyle = BS_PATTERN;
  MyLogBrush.lbHatch = HMyBit;
  TheBrush := CreateBrushIndirect(&MyLogBrush);
}
```

Zum Testen des Musters wird es in einem Rechteck angezeigt:

```
void MyWindow::Paint(HDC PaintDC, PAINTSTRUCT& PaintInfo)
{
  SelectObject(PaintDC, TheBrush);
  Rectangle(PaintDC, 20, 20, 200, 200);
}
```

Bitmap und Pinsel müssen nach Gebrauch des Pinsels gelöscht werden:

```
DeleteObject(HMyBit);
DeleteObject(TheBrush);
```

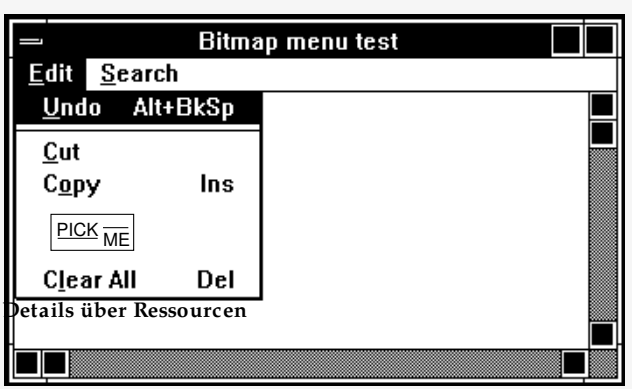
Bitmaps in Menüs anzeigen

Die Anzeige eines Bitmaps in einem Menü erfolgt mit der Funktion *ModifyMenu*. Ein vorhandener Menüpunkt wird dabei so verändert, daß anstelle des Menütexts, der für den Menüpunkt in einem Menüeditor definiert wurde, das Bitmap angezeigt wird. Im folgenden Beispiel verändert der Konstruktor ein Fenstermenü:

```
SampleWindow::SampleWindow(PTWindowsObject AParent LPSTR
                           ATitle) : TWindow(AParent, ATitle)

  WORD HMyBit; {
AssignMenu(100);
  HMyBit = LoadBitmap(GetApplication()->HInstance,
                      MAKEINTRESOURCE(503));
  ModifyMenu(GetMenu(HWindow), 111, MF_BYCOMMAND | MF_BITMAP,
             211, (LPSTR)HMyBit);
}
```

In diesem Beispiel ist 111 die Befehls-ID der Menüauswahl, die geändert werden soll, 211 ist die neue ID. Es könnte aber auch dieselbe ID verwendet werden.



21. Richtlinien für Standardanwendungen

Für die Entwicklung von Windows-Anwendungen sprechen vor allem die leichte Erlernbarkeit und der einfache Gebrauch dieser Programme. Zu der standardisierten Oberfläche der Anwendungen gehört es, daß Operationen, die allen Programmen gemeinsam sind, auf dieselbe Weise angeboten werden. So hat zum Beispiel das Edit-Menü sowohl in Text-, als auch in Grafik- und Zeichenanwendungen immer dieselben Menüpunkte und Tastenkürzel. Mit anderen Worten: Wenn Sie mit einer Windows-Anwendung umgehen können, können Sie mit allen anderen auch umgehen.

Um die Richtlinien für den Zugriff auf gemeinsame Operationen zu formalisieren, hat IBM Regeln aufgestellt, die unter dem Begriff *Common User Access* (CUA) zusammengefaßt sind. Obwohl diese Regeln für OS/2 Presentation-Manager-Anwendungen aufgestellt wurden, haben sie auch für Windows-Anwendungen Gültigkeit und werden generell als Industriestandard angesehen. Dieses Kapitel beruht auf den von IBM veröffentlichten CUA-Richtlinien.

Gestaltungsprinzipien

Für Anwendungen, die sich an die CUA-Richtlinien halten, gibt es zwei vornehmliche Ziele: Der Anwender soll die Wirkung jeder Anwendungsaktion sicher einschätzen und den Fluß der Ereignisse steuern können.

Antworten, die der Erwartung des Anwenders entsprechen

Wenn eine Anwendung konsistente und der aktuellen Situation entsprechende Antworten gibt, stellt sich beim Anwender schneller eine gewisse Gewöhnung ein. Überdenken Sie in dieser Hinsicht die folgenden Vorschläge:

Stellen Sie Analogien her. Gestalten Sie das Verhalten und die Eigenschaften einer Anwendung so, daß sie auf Grundlagen basiert, die dem Anwender vertraut und bekannt sind. Entwerfen Sie Ihre Anwendung also analog zu einem bereits bestehenden, nicht »computerisierten« Verfahren. Der Anwender wird so sehr

viel schneller und leichter das der Anwendung zugrundeliegende Konzept erfassen und akzeptieren.

Achten Sie auf Konsistenz. Achten Sie auf die Konsistenz der Anwendung in Bezug auf Erscheinungsbild und Gebrauch. Verwenden Sie gleiche Befehle (Auswahl aus Menüs, Anklicken von Symbolen, etc.) für gleiche Operationen. Gestalten Sie Bildschirme für ähnliche Aufgaben auch in ähnlicher Weise.

Vermeiden Sie Modi. Ein Modus ist ein Zustand der Anwendung, der den Anwender-Zugriff auf bestimmte Eigenschaften beschränkt. Modi werden oft dazu mißbraucht, um die Aktivitäten des Anwenders in eine ganz bestimmte Richtung zu lenken. Modi sind bei kurzen Fehlermeldungen und bei der Verwendung von Werkzeugen durchaus angebracht. Wenn der Mauszeiger zum Beispiel ein Stift ist, kann der Anwender zeichnen, aber nicht Formen verschieben. Sorgen Sie für eine optische Darstellung der Modi, zum Beispiel über wechselnde Darstellungen des Mauszeigers oder grau hinterlegte Menüauswahlpunkte.

Steuerung des Programmflusses durch den Anwender

Der Anwender fühlt sich sicherer, wenn er den Ablauf der Aktivitäten steuern kann. Die Kontrolle über die Arbeitsschritte läßt die Aura des Geheimnisses gar nicht erst aufkommen, die ja Computerprogramme immer noch umgeben soll. Das Programm soll schließlich zu einem wirkungsvollen Mittel zum Zweck werden.

Listen Sie zur Auswahl stehende Möglichkeiten auf. Stellen Sie dem Anwender eine Liste zur Verfügung, die die in Frage kommenden Aktionen enthält. Normalerweise beinhalten Elementfunktionen eine Palette von Werkzeugen oder Farben und ein Menü mit Optionen.

Geben Sie Rückmeldungen. Jede Anwender-Aktion sollte eine optische oder akustische Rückmeldung zur Folge haben.

Unterstützen Sie das Experimentieren mit der Anwendung. Hindern Sie Anwender nicht am Ausprobieren von Möglichkeiten. Das Experimentieren ist häufig der beste Weg zum Kennenlernen einer neuen Anwendung. Bauen Sie unbedingt eine Widerrufen-Option in die Anwendung ein.

Standards für Erscheinungsbild und Verhalten

Der Programm-Manager und die Beispielprogramme zeigen bereits, wie Erscheinungsbild und Verhalten einer Anwendung aussehen sollten. So hat z.B. jede Anwendung ein Hauptfenster. Einfache und doppelte Mausklicks haben in jeder Anwendung ähnliche Resultate.

Generelle Präsentation

Halten Sie sich an die Abfolge Objekt – Aktion. Folgende Abfolge von Ereignissen ist typisch für eine Anwendungsoperation: Der Anwender wählt einen Eintrag oder ein Objekt aus und führt eine Aktion damit aus. So kann der Anwender zum Beispiel in einem Textprogramm Text auswählen oder markieren und dann die Schrift oder die Größe ändern. In einem Zeichenprogramm wählt der Anwender zum Beispiel ein Rechteck aus und verschiebt es an eine andere Position auf dem Bildschirm. Obwohl es eine Vielzahl von Möglichkeiten für die Auswahl von Objekten gibt und womöglich eine noch größere Anzahl von verschiedenen Möglichkeiten für die Auswahl von Aktionen, sollten Sie dennoch versuchen, sich an dieses Grundmodell zu halten.

Wählen Sie Ihre Fenster mit Bedacht. Eine Windows-Anwendung kann in einem oder mehreren Fenstern angezeigt werden. Damit Ihre Anwendung von optimalem Nutzen ist, müssen Sie sorgfältig mit den verschiedenen Fenstertypen und ihren Einsatzmöglichkeiten umgehen. Das *Hauptfenster* sollte als Basis für die Anwendung dienen, sozusagen ihr »Rückgrat« sein. Es repräsentiert die gesamte Anwendung dadurch, daß seine Verkleinerung zu einem Symbol sämtliche Aktionen des Programms aussetzt. Der Anwender kann das Hauptfenster aber auch so vergrößern, daß es den gesamten Bildschirm einnimmt. *Sekundäre* Fenster, wie Pop-up- oder Dialogfenster, sollten vom Hauptfenster als Ergebnis einer Aktion im Hauptfenster eingeblendet werden. Pop-up-Fenster können ein separates Modul der Anwendung repräsentieren, sollten aber nicht zu einem Symbol verkleinert werden.

Maus- und Tastatur-Interaktionen

Zeigen Sie die Auswahl eines Objekts an. Wenn ein Objekt durch einen Mausklick ausgewählt wurde, sollten Sie dies optisch darstellen, indem Rechtecke an den Ecken des gewählten Objekts eingeblendet oder Farbe und/oder Stärke geändert werden. Diese Auswahlmarkierung darf aber nicht dauerhaft mit dem Objekt verbunden bleiben. Wenn der Anwender eine Aktion mit dem Objekt durchführt, muß die Markierung beibehalten werden. Der Anwender kann dann weitere Aktionen durchführen, die das Objekt betreffen. Deaktivieren Sie die Auswahlmarkierung, wenn der Anwender ein anderes Objekt auswählt.

Schränken Sie die Bewegungsfreiheit des Mauszeigers nicht ein. Der Mauszeiger sollte von einem Fenster zum anderen und von einer Anwendung zur anderen bewegt werden können. Bewegen Sie den Mauszeiger nicht unabhängig von der Maus: dadurch würde die unmittelbare Verbindung zwischen Mauszeiger und Maus unterbrochen.

Verwenden Sie den Tastaturcursor. Dieser Cursor markiert den Punkt, an dem Eingaben über die Tastatur erfolgen werden. So stellt das blinkende Caret den Tastaturcursor in Textverarbeitungs-Anwendungen dar. Er existiert unabhängig

vom Mauszeiger, kann aber durch Anklicken an die Position des Mauszeigers versetzt werden.

Markieren Sie den Fokus. Der *Fokus* ist die Stelle am Bildschirm, die Eingaben über die Tastatur oder die Maus aufnimmt. Wenn der Anwender zum Beispiel ein Pop-up-Menü auswählt, besitzt dieses den Fokus. Wenn der Anwender Text optisch hervorhebt, hat dieser Text den Fokus. Markieren Sie immer das fokussierte Objekt: Ausgewählte Menüpunkte werden normalerweise invers dargestellt und ausgewählte Steuerelemente in einem Dialogfenster mit einer gepunktete Linie umgeben. Wenn der Anwender die Maus zieht, um eine Gruppe von Einträgen auszuwählen, wird der gesamte Bereich häufig mit einer gepunkteten Linie umgeben, die »Marquee« genannt wird.

Entwerfen Sie eine Tastaturschnittstelle. Stellen Sie mit den Pfeiltasten, der Tab-Taste und Tastenkürzeln Alternativen zur Maus zur Verfügung. Das ist wichtig für Anwender, die keine Maus haben oder sie nicht verwenden wollen.

Halten Sie sich an die Konventionen für das Klicken mit der Maus. Windows verwendet bestimmte Konventionen für das Klicken mit der Maus, die jedoch nicht zwingend sind. So bewirkt das Drücken der linken Maustaste die Auswahl und Manipulation von Objekten, das Drücken der rechten Taste führt andere Aufgaben aus, wie zum Beispiel Ändern des Modus oder Anzeigen einer Pop-up-Liste mit Optionen.

Für die linke Maustaste gelten folgende Regeln:

- Ein einfacher Klick wählt ein Objekt aus.
- Ein Doppelklick wählt ein Objekt aus und aktiviert eine damit verbundene, voreingestellte oder implizierte Aktion.
- Klicken und Ziehen verändert die Größe und Position des Objekts oder wählt Text aus und markiert ihn.
- Strg -Taste und einfacher Klick fügt das ausgewählte Objekt zu den vorher ausgewählten Objekten hinzu.
- Umschalt -Taste und einfacher Klick erweitert den Bereich vorher ausgewählter Objekte zwischen dem als Anker fungierenden ersten und dem angeklickten Objekt (vgl. dazu beispielsweise den Datei-Manager).

Menüs

Windows unterstützt und erzwingt einen festumrissenen Standard für das Erscheinungsbild von Pull-down-Menüs. Trotzdem können Sie über die ausgewählten Stile und die Optionen, die darin angeboten werden sollen, relativ frei entscheiden. So sollten Sie zum Beispiel eindeutige Abkürzungen für die Tasten bzw. Tastenkombinationen vergeben, mit denen der Anwender auf Einträge in der obersten Menüzeile und in Pull-down-Menüs zugreifen kann. Einen Menüeintrag, der nach seiner Auswahl ein Dialogfenster einblendet, sollten Sie mit nachfolgenden ... markieren. Hinterlegen Sie Menüeinträge, die nicht aktiviert werden können, grau.

Die meisten Anwendungen – gleichgültig für welche Zwecke sie entworfen sind – führen immer bestimmte Operationen zur Manipulation von Dateien, zum Editieren und zum Zugriff auf ein Hilfesystem aus. Aus diesem Grund definiert CUA besondere Formate für Datei-, Edit- und Hilfemenüs. Außerdem gibt es Richtlinien für Anzeige- und Optionenmenüs.

Gemäß CUA sind die Optionen des Hauptmenüs in der folgenden Reihenfolge aufgelistet, wobei das Hilfemenü immer an letzter Stelle (das heißt, ganz rechts) steht, unabhängig davon, wieviele andere Menüauswahlpunkte angeboten werden.

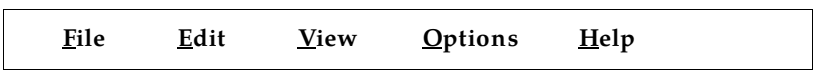


Abbildung 21.1 Eine CUA-gemäße Menüzeile

Dialogfenster

Den vielen Möglichkeiten zum Entwurf von Dialogfenstern stehen nur wenige Richtlinien gegenüber. Sie sollten sich für eine Möglichkeit entscheiden und diese dann »durchhalten«. Es gibt zwei Typen von Dialogfenstern: modale und nicht-modale. Am häufigsten werden modale Dialoge verwendet, die die Operation des Programms sozusagen einfrieren. Nicht-modale Dialoge, die die reguläre Fortsetzung des Programms erlauben, sollten Sie nur für ganz spezielle Zwecke und parallel laufende Aktivitäten einsetzen, wie zum Beispiel dann, wenn ein Dokument auf Rechtschreibfehler überprüft wird.

Für Öffnen- und Sichern-Als-Dialoge definiert CUA eine besondere Art von Dialogfenster. Das in ObjectWindows vordefinierte Datei-Dialogfenster entspricht dieser Definition.

Weitere Anmerkungen zur Gestaltung

Wählen Sie die richtige Größe für das Hauptfenster. Das Hauptfenster sollte gerade so groß sein, daß die notwendigen Informationen darin Platz haben und daß ein ausreichend großer Client-Bereich angelegt werden kann.

Seien Sie sorgfältig bei der Veränderung von Fenstergrößen. Es gibt zwei Möglichkeiten für das Neuzeichnen von Fenstern, deren Größe verändert wurde.

- Sie können die Fensterinhalte in den ursprünglichen Proportionen anzeigen, wobei die Bereiche, die nicht mehr in das Fenster passen, abgeschnitten werden. Dieses Vorgehen eignet sich für sogenannte »what-you-see-is-what-you-get«-Anwendungen (WYSIWYG).

- Alternativ dazu kann der Inhalt aber neu formatiert werden, so daß er genau in das Fenster paßt. Diese Möglichkeit ist für Spiele und Utilities angebracht.

Verwenden Sie Farben sparsam. Farben sollten eine bestehende Kommunikationsverbindung mit dem Anwender fördern. Die Farbgebung sollte konsistent sein und der Anwender muß die Möglichkeit zur Änderung der einzelnen Farben haben.

Sichere Programme

Die Fehlerbehandlung in Programmen mit interaktiver Anwenderschnittstelle ist weitaus komplizierter als in Kommandozeilen-Anwendungen. In einer nicht-interaktiven Anwendung gilt es als ganz normal (und wird sogar erwartet), daß Fehler zur Anzeige einer Meldung und zur Beendigung des Programms führen. Ein interaktives Programm jedoch, muß Fehler abfangen und in einem definierten Zustand weiterarbeiten. Das Programm darf unter keinen Umständen die Daten zerstören, die der Anwender gerade bearbeitet oder einfach abbrechen. Ein Programm, das diese Kriterien erfüllt, kann als »sicher« bezeichnet werden.

ObjectWindows erleichtert das Schreiben von sicheren Programmen. Es unterstützt einen Programmierstil, der das Entdecken und Beheben von Fehlern leichter macht, vor allem die schwer in den Griff zu bekommenden Fehler, die von ungenügenden Arbeitsspeicher herrühren. Das Konzept der atomaren Operationen macht diese Art der Fehlerbehandlung möglich.

Programmieren unter dem Motto »Alles oder Nichts«

Eine atomare Operation ist eine Operation, die sich nicht weiter in kleinere Operationseinheiten zerlegen läßt. In unserem Zusammenhang heißt das: Eine atomare Operation schlägt entweder vollkommen fehl oder ist hundertprozentig erfolgreich. Atomare Operationen sind vor allem bei der Zuweisung von Speicherplatz sehr hilfreich.

Programme reservieren Speicherplatz normalerweise in kleinen Portionen. Wenn Sie zum Beispiel ein Dialogobjekt erstellen, wird für das Dialogfenster und für alle anderen zum Dialogfenster gehörigen Objekte (wie untergeordnete Steuerelemente) Speicherplatz reserviert. Jede dieser Reservierungen könnte potentiell fehlschlagen. Deshalb muß nach jeder Speicherplatzzuweisung überprüft werden, ob die nächste Reservierung durchgeführt werden kann oder nicht. Wenn eine Reservierung nicht erfolgreich war, muß bereits belegter Speicherplatz wieder freigegeben werden. Es wäre ideal, wenn Sie alle Reservierungen durchführen und dann überprüfen könnten, ob eine davon mißlungen ist. Hier kommt der *Sicherheitsbereich* ins Spiel.

Der Sicherheitsbereich

ObjectWindows reserviert einen Speicherbereich, der *Sicherheitsbereich* genannt wird. Die Konstante *DEFAULT_SAFETY_POOL_SIZE* in safepool.h defi-

niert die voreingestellte Größe des Sicherheitsbereichs. Wenn das Reservieren von Speicherplatz im Heap den Sicherheitsbereich berührt, liefert die Elementfunktion *LowMemory* des Anwendungsobjekts den Wert *TRUE* zurück. Das bedeutet, daß weitere Reservierungen nicht sicher sind und unter Umständen fehlschlagen können.

Damit der Sicherheitsbereich seine Aufgabe erfüllen kann, muß er mindestens so groß sein, wie die größte atomare Zuweisung. Mit anderen Worten: Der Bereich muß groß genug sein, daß alle Reservierungen zwischen den Überprüfungen mit *LowMemory* erfolgreich sind; normalerweise reicht dafür die voreingestellte Größe des Sicherheitsbereichs aus.

Wenn Ihre Anwendung zum ersten Mal aktiv wird und das Haupt- und die untergeordneten Fenster erstellt, überprüft sie auch gleich, ob genügend Speicherplatz zur Verfügung steht. Wenn das Programm jedoch nach der Anzeige des Hauptfensters Fenster oder Dialoge als dynamische Objekte erstellt, müssen Sie explizit den Mechanismus zur Speicherplatz-Überprüfung aufrufen. Dies ist erforderlich, wenn Dialoge oder Pop-up-Fenster als Antwort auf eine Menüauswahl erzeugt werden. MDI-Fenster überprüfen bei der Erstellung von untergeordneten Fenstern von vorneherein, ob genügend Speicherplatz zur Verfügung steht.

Rufen Sie die *Create*-Elementfunktion Ihrer Dialogfensterobjekte nicht direkt auf, da *Create* ein Fenster erzeugt, ohne vorher zu überprüfen, ob genügend Speicherplatz verfügbar ist:

```
void TMywindow::CMHelp(RTMessage Msg)
{
  PTHelpWindow HelpWindow;

  HelpWnd = new THelpWindow(this, "Help");
  HelpWnd->Create;
}
```

Die folgende Elementfunktion überprüft bei der Fenstererstellung, ob genügend Speicherplatz zur Verfügung steht:

```
void TMyWindow::CMHelp(RTMessage Msg)
{
  PTHelpWindow HelpWindow;

  HelpWindow = new THelpWindow(this, "Help");
  if ( GetApplication()->MakeWindow(HelpWindow) )
    // Das Fenster wurde erfolgreich erstellt
  ...
}
```

MakeWindow liefert entweder einen Zeiger auf das Fensterobjekt oder *NULL* zurück. Letzteres ist der Fall, wenn die Erstellung nicht erfolgreich ausgeführt werden konnte.

Wenn Sie bei der Ausführung von modalen Dialogen überprüfen wollen, ob genügend Speicherplatz vorhanden ist, so rufen Sie *Application*()->*ExecDialog* auf:

```
void TMyWindow::CMRunDialog(RTMessage Msg)
{
  PTTestDialog TestDialog;
  int RetValue;

  TestDialog = new TTestDialog(this, "Test Dialog");
  RetValue = GetApplication()->ExecDialog(TestDialog);
  if ( RetValue == IDCANCEL )
    // Dialog wurde nicht erstellt oder vom Benutzer
    // abgebrochen
  ...
}
```

Bei erfolgreicher Durchführung liefert *ExecDialog* den Rückgabewert des Dialogs, normalerweise Null oder einen positiven Integerwert. Wenn die Ausführung des Dialogs nicht erfolgreich war, liefert *ExecDialog* eine Fehlerkonstante in Form eines negativen Integerwerts.

MakeWindow und *ExecDialog* benachrichtigen das Anwendungsobjekt, wenn zu wenig Speicherplatz vorhanden ist. Dazu wird die *Error*-Elementfunktion des Anwendungsobjekts mit der Konstante *EM_OUTOFMEMORY* als Argument aufgerufen. Die Elementfunktion *Error*, die von *TModule* definiert wird, zeigt ein Meldungsfenster an, das den Anwender auf die Fehlersituation aufmerksam macht. ObjectWindows überschreibt den globalen Operator delete, damit er den Sicherheitsbereich nach jedem Löschen überprüft. Ist der Sicherheitsbereich aufgebraucht, versucht der Operator delete, ihn neu zu reservieren. Der Sicherheitsbereich wird also sofort wieder hergestellt, wenn dies möglich ist. Weitere Informationen über den Sicherheitsbereich finden Sie in Kapitel 24 »Weitere Komponenten von ObjectWindows«.

Weitere Fehler bei der Erstellung von Fenstern

Neben nicht ausreichendem Speicherplatz gibt es weitere Bedingungen, die die Erstellung und korrekte Anzeige eines Fensters oder eines Dialogs verhindern. Microsoft Windows läßt die Erstellung eines Schnittstellenelements nicht zu, wenn nicht ausreichend Speicherplatz für Windows selbst zur Verfügung steht oder wenn ungültige Erstellungsattribute verwendet werden.

Die ObjectWindows-Elementfunktionen, die Schnittstellenobjekte initialisieren und erstellen, melden die oben erwähnten Fehler. *TWindowsObject* definiert das int-Datenelement *Status*, dessen Wert ungleich Null ist, wenn ein Fehler auftritt. Tabelle 21.1 enthält einige der Fehlerkonstanten von ObjectWindows.

Eine vollständige Liste mit den Fehlerkonstanten und ihren Werten finden Sie unter »EM_XXXX-Konstanten« in Kapitel 24 »Weitere Komponenten von ObjectWindows« dieses Handbuchs.

Wenn Sie überprüfen wollen, ob die Fenster-Konstruktoren und -Erstellungselementfunktionen Fehler enthalten, besetzen Sie *Status* mit einer der oben genannten oder einer eigenen Konstanten. Wenn ein Fenster zum Beispiel eine

Datei öffnen muß, dabei aber nicht erfolgreich ist, kann es den Fehler mit Hilfe einer Konstante *EM_FILEOPENERROR* melden.

Wenn *TWindow::Create* nach einem Aufruf durch *MakeWindow FALSE* zurückliefert oder *TDialog::Execute Status* nach einem Aufruf durch *ExecDialog* auf einen Wert ungleich Null setzt, meldet die aufrufende Elementfunktion den Fehler an die *Error*-Elementfunktion des Fenstertyps. Diese ruft per Voreinstellung die *Error*-Elementfunktion der Anwendungsklasse auf. Sie können die *Error*-Elementfunktion der Fenster- oder Anwendungsklasse aber überschreiben, wenn Sie jeden Fehler gesondert behandeln möchten:

```
TEditorWindow::TEditorWindow(PTWindowsObject AParent, LPSTR
                             ATitle, LPSTR AFileName):
                             TWindow(AParent, ATitle)
{
  if ((Status == 0) && !OpenTheFile(AFileName))
    Status = EM_FILEOPENERROR;
}
```

Tabelle 21.1 Werte für Fehler bei der Fenstererstellung

21 **Sichere Programme** 309

EM_OUTOFMEMORY	Die Speicherreservierung hat den Sicherheitsbereich berührt.
EM_INVALIDWINDOW	Das Fensterelement konnte nicht erstellt werden.
EM_INVALIDCLIENT	Ein MDI-Fensterobjekt konnte sein Client-Fenster nicht erstellen.
EM_INVALIDCHILD	Ein oder mehrere untergeordnete Fenster des Objekts haben einen Fehler erzeugt.
EM_INVALIDMAINWINDOW	Das Hauptfenster konnte nicht angelegt werden.

Großverbraucher

Sie sollten auch überprüfen, ob genügend Speicherplatz für die Großverbraucher zur Verfügung steht. Großverbraucher sind z.B. Fensterobjekte, die Speicherplatz reservieren, dessen Umfang größer ist als der Sicherheitsbereich oder Objekte, die den gesamten Inhalt einer Datei in den Speicher laden.

Wenn ein Großverbraucher keinen Speicherplatz mehr zur Verfügung hat, während er erstellt wird, sollte er sein *Status*-Datenelement mit einem Fehlerwert besetzen und nicht versuchen, weiteren Speicherplatz zu reservieren. Das *Status*-Datenelement wird dann in der *MakeWindow*-Elementfunktion überprüft, der Fehler gemeldet und das Fensterobjekt freigegeben.

Dieses Vorgehen ist natürlich mit einem gewissen Aufwand verbunden. Da aber nicht immer davon ausgegangen werden kann, daß ausreichend Speicherplatz zur Verfügung steht, handelt es sich um die einzige sichere Möglichkeit, Objekte zu erstellen, die größer als der Sicherheitsbereich sind.

IV. ObjectWindows-Referenz

22. ObjectWindows-Klassen

Dieses Kapitel enthält eine alphabetische Aufzählung aller ObjectWindows-Standardklassen. Zu jeder Klasse finden Sie eine Erklärung ihrer Aufgabe, Verwendung, Elemente, Operatoren und *Friends*.

Um Informationen über eine bestimmte Klasse zu finden, sollten Sie bedenken, daß viele Eigenschaften eines Objekts geerbt sind. Um diese Informationen nicht endlos zu vervielfachen, dokumentiert das vorliegende Kapitel nur Datenelemente und Elementfunktionen, die für eine bestimmte Klasse *neu* oder *verändert* sind.

Wenn Sie Informationen über ein bestimmtes vererbtes Element suchen, können Sie entweder im Index oder in den Basisklassen nachsehen.

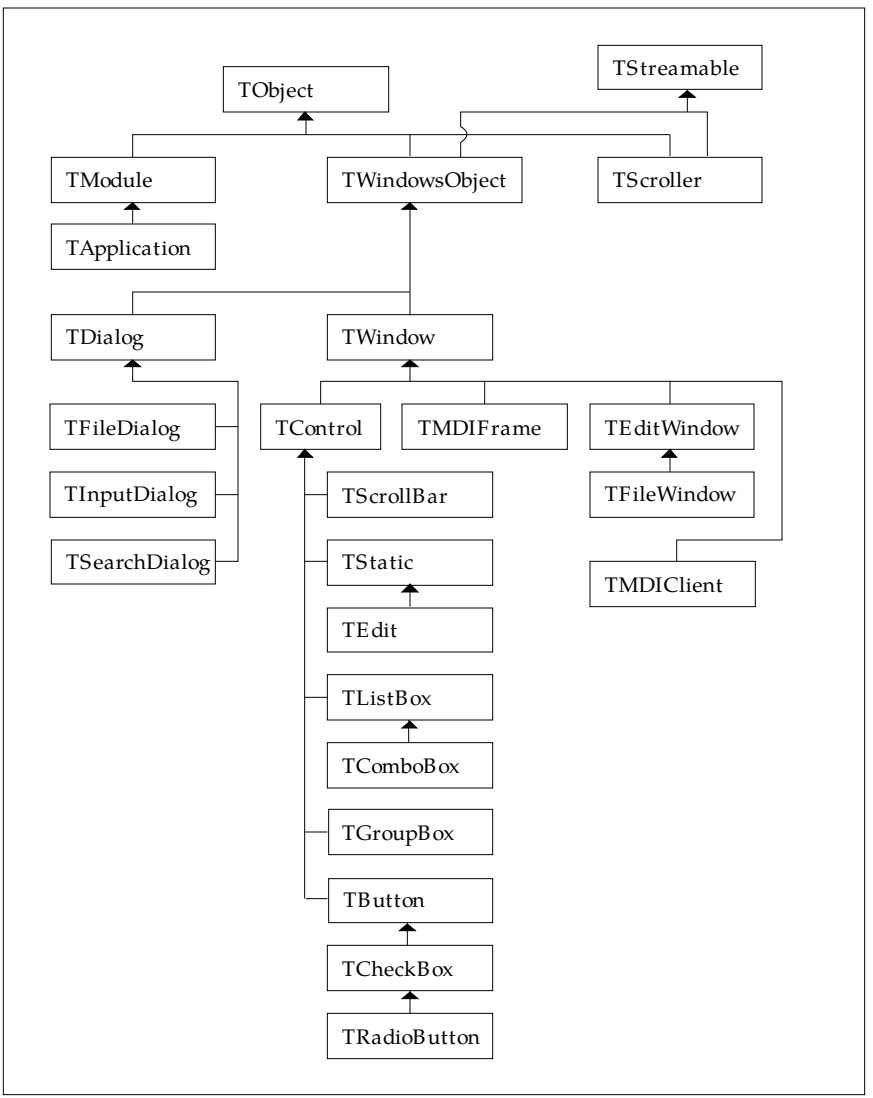
Wenn Sie z.B. etwas über das Datenelement *Parent* der Klasse *TEdit* wissen möchten, gehen Sie folgendermaßen vor:

- Zuerst sehen Sie bei den Datenelementen von *TEdit* nach, wo aber *Parent* nicht aufgeführt ist.
- Danach suchen Sie *TStatic*, die Basisklasse von *TEdit* in der Hierarchie. *Parent* ist hier ebenfalls nicht aufgelistet.
- Also überprüfen Sie als nächstes *TControl*, die Basisklasse von *TStatic*.
- Dieses Vorgehen setzen Sie solange fort, bis Sie schließlich zu *TWindowsObject* gelangen. Hier finden Sie die vollständige Information über *Parent*, das *TEdit* unverändert von *TWindowsObject* erbt (die folgende Abbildung 22.1 zeigt die Hierarchie).

Das vorliegende Kapitel ist eine Referenz dieser Klassen.

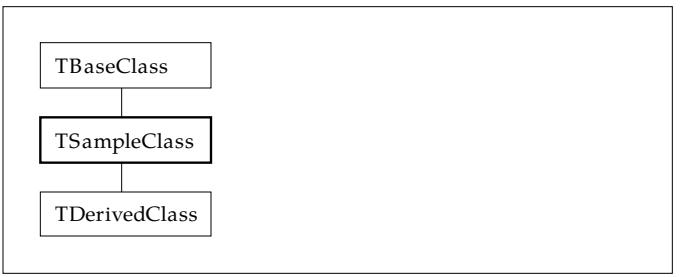
Hinweis: Die Klasse *TStreamable* ist in Kapitel 23 »Stream-fähige Klassen in ObjectWindows« erläutert.

Der Eintrag für jede Klasse hat das unter »TSampleClass« beschriebene Format.


TObject
TStreamable
TModule
TWindowsObject
TScroller
TApplication
TDialog
TWindow
TFileDialog
TControl
TMDIFrame
TEditWindow
TInputDialog
TScrollBar
TFileWindow
TSearchDialog
TStatic
TMDIClient
TEdit
TListBox
TComboBox
TGroupBox
TButton
TCheckBox
TRadioButton

TSampleClass

Header-Datei der Klasse



In diesem Abschnitt finden Sie einen allgemeinen Überblick über die Klasse, ihre Beziehung zu anderen Klassen und ihre allgemeine Verwendung. Danach folgt eine Auflistung der Datenelemente und Elementfunktionen.

Weitere Informationen über ObjectWindows-Typen oder Konstanten (wie z.B. *EM_OUTOFMEMORY*) finden Sie in Kapitel 24 »Weitere Komponenten von ObjectWindows«. Windows-Typen (wie *LPSTR*) sind in Kapitel 3 »Windows-Typen« im *Windows-Referenzhandbuch* erläutert.

Wenn eine Funktion als protected deklariert ist, so wird dies rechts neben dem Prototyp angegeben. Das Zugriffsattribut public wird nicht besonders vermerkt und private Elemente werden nicht aufgeführt.

Datenelemente

Dieser Abschnitt beschreibt alle neu hinzugekommenen Datenelemente der Klasse in alphabetischer Reihenfolge. Sie finden hier die Elementdeklaration, das Zugriffsattribut (wenn es nicht public ist) und eine Beschreibung des Zwecks.

ADataMember

```
SomeType ADataMember;
```

ADataMember ist ein Datenelement der Klasse. Es wird erklärt, was es enthält und wie man es anwendet.

Querverweis: Verwandte Datenelemente, Elementfunktionen, Klassen, globale Funktionen, usw.

AnotherDataMember

```
AnotherType AnotherDataMember;
```

protected

AnotherDataMember enthält ähnliche Informationen wie *ADataMember*.

Elementfunktionen

Dieser Abschnitt listet alle Elementfunktionen auf, die für die Klasse entweder neu definiert sind oder die geerbte Elementfunktionen überschreiben.

Konstruktor

```
TSampleClassName(SomeType AParameter);
```

Konstruktor für eine neue Klasse; setzt das Datenelement *ADataMember* auf *AParameter*.

Zilch

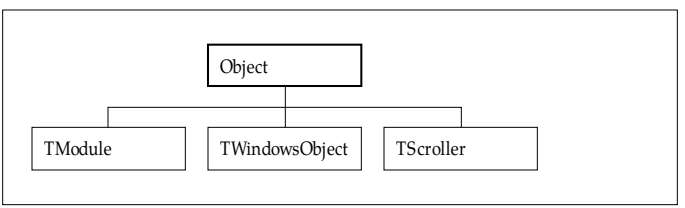
```
virtual void Zilch();
```

Die Elementfunktion *Zilch* veranlaßt die Klasse dazu, eine Aktion auszuführen.

Querverweis: *TSomethingElse::Zilch*

Object

object.h



Object ist der Ausgangspunkt der ObjectWindows-Hierarchie. *Object* ist eine abstrakte Klasse, die das Verhalten und die Eigenschaften der abgeleiteten Objekte bestimmt. Sie stellt den Basismechanismus und die Basisstruktur für die Typenprüfung und die Kapselung zur Verfügung.

Datenelemente

ZERO

```
static PObject ZERO;
```

Zeiger auf ein Fehlerobjekt, der von einer privaten Klasse verwendet wird. Er behandelt auftretende Probleme, wenn der Operator *new* für ein Objekt keinen Speicherplatz reservieren kann.

Elementfunktionen

Konstruktor

```
Object();
```

Erstellt eine Instanz von *Object*.

Konstruktor

```
Object(RObject);
```

Kopier-Konstruktor. Kopiert Datenelemente in eine andere Instanz von *Object*.

Destruktor

```
virtual ~Object();
```

Virtueller Destruktor. Gibt eine Instanz von *Object* frei.

firstThat

```
virtual RCObject firstThat(condFuncType, Pvoid) const;
```

Ruft die angegebene Testfunktion für ein Objekt mit einer Parameter-Liste auf und ermittelt das erste Objekt in der Container-Klasse, das die Bedingungen erfüllt. Liefert das Objekt zurück, wenn der Test erfolgreich war, andernfalls NOOBJECT.

forEach
```
virtual void forEach(iterFuncType, Pvoid);
```
Ruft die gegebene Funktion mit der gegebenen Parameterliste für alle Objekte der Container-Klasse auf.

hashValue
```
virtual hashValueType hashValue() const = 0;
```
Abstrakte Funktion, die einen Hash-Wert – einen eindeutigen Schlüssel, der auf dem Objekt basiert – zurückliefert.

isA
```
virtual classType isA() const = 0;
```
Abstrakte Funktion, die einen eindeutigen Klassenbezeichner für die Klasse zurückliefert.

isAssociation
```
virtual int isAssociation() const;
```
Stellt fest, ob das Objekt von der Klasse *Association* abgeleitet ist. Liefert 0 für die Basisklasse *Object* zurück.

isEqual
```
virtual int isEqual(const RCObject) const = 0;
```
Abstrakte Funktion, die testet, ob zwei Objekte vom selben Typ und ihre Datenelemente gleich sind.

isSortable
```
virtual int isSortable() const;
```
Stellt fest, ob das Objekt sortiert werden kann; das ist dann der Fall, wenn das Objekt von *Sortable* abgeleitet ist. Liefert für die Basisklasse *Object* 0 zurück, weil diese nicht sortiert werden kann.

lastThat
```
virtual RCObject lastThat (condFuncType, Pvoid) const;
```
Dieselbe Funktion wie *firstThat* für Nicht-Container-Objekte. Sie wird für Container-Objekte überschrieben, um eine Referenz auf das letzte Objekt zurückzuliefern, bei dem die gegebene Bedingungsfunktion 1 zurückliefert.

nameOf
```
virtual Pchar nameOf() const = 0;
```
Abstrakte Funktion, die einen Bezeichnungsstring für eine Klasse zurückliefert.

new
```
Pvoid operator new(size_t);
```
Reserviert für ein Objekt eine gegebene Anzahl von Bytes. Wenn die Reservierung nicht erfolgreich war, wird *ZERO* zurückgeliefert.

printOn
```
virtual void printOn(Rostream) const = 0;
```
Abstrakte Funktion, die eine druckbare Form eines Objekts in einen Stream schreibt.

Friend

operator <<
```
Rostream operator <<(Rostream, RCObject);
```
Friend-Funktion, die ein Objekt in einen Ausgabe-Stream schreibt.

Die Operatoren >> und <<

Für alle stream-fähigen Klassen werden vier Operatoren als Funktionen definiert. Zu diesen Klassen gehören *TButton, TCheckBox, TComboBox, TDialog, TEdit, TEditWindow, TFileDialog, TFileWindow, TInputDialog, TGroupBox, TListBox, TMDIClient, TMDIFrame, TRadioButton, TScrollBar, TScroller, TStatic, TWindow* und *TWindowsObject*. *TClassName* bezieht sich also auf eine dieser Klassen.

Es gibt jeweils zwei << und >>Operatoren. Diese Operatoren sind zwar meist nicht erforderlich, sie stellen jedoch sicher, daß es bei Mehrfachvererbung nicht zu Mehrdeutigkeiten kommen kann.

Beachten Sie den Unterschied zwischen den beiden *operator* >> Funktionen: die erste übernimmt eine *Referenz* auf ein *TClassName*-Objekt, die zweite eine *Referenz* auf einen Zeiger auf ein *TClassName*-Objekt als Parameter (Operand). Derselbe Unterschied besteht zwischen den beiden <<-Operatoren.

operator >>

```
Ripstream operator >> (Ripstream is, RTClassName cl);
Ripstream operator >> (Ripstream is, RPTClassName cl);
```

Liest ein *TClassName*-Objekt aus dem Eingabe-Stream *is* und schreibt es in *cl*. Es wird eine Referenz auf den Stream zurückgeliefert, was die Verkettung von >>Operatoren erlaubt.

Querverweis: *ipstream*

operator <<

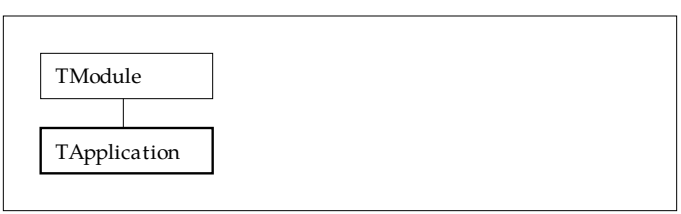
```
Ropstream operator << (Ropstream os, RTClassName cl );
Ropstream operator << (Ropstream os, PTClassName cl );
```

Schreibt das *TClassName*-Objekt in einen Ausgabe-Stream *os*. Es wird eine Referenz auf den Stream zurückgeliefert, was die Verknüpfung von <<-Operatoren erlaubt.

Querverweis: *opstream*

TApplication

applicat.h



TApplication ist von *TModule* abgeleitet und fungiert als objektorientierte Repräsentanz für Windows-Anwendungsmodule. *TApplication* und *TModule* implementieren zusammen das grundlegende Verhalten und die Eigenschaften für

eine Windows-Anwendung. *TApplication*-Elementfunktionen führen die Instanzen-Initialisierung aus, einschließlich Erstellung des Hauptfensters und Botschaftsbearbeitung.

Datenelemente

HAccTable

```
HANDLE HAccTable;
```

HAccTable enthält ein Handle für die aktuell von der Anwendung eingesetzte Windows-Tastenkürzeltabelle.

hPrevInstance

```
HANDLE hPrevInstance;
```

Enthält ein Handle für die schon bestehende Instanz der Windows-Anwendung. Wenn *hPrevInstance* 0 ist, gibt es keine solche Instanz.

KBHandler-Wnd

```
PTWindowsObject KBHandlerWnd;
```

KBHandlerWnd zeigt auf das aktive Fenster, wenn dieses Tastatureingaben entgegennimmt. Damit kann ein Fenster mit Steuerelementen Tastatureingaben bearbeiten. Dabei wird der Fokus von Steuerelement zum Steuerelement weitergegeben, wenn die Tab -Taste und/oder die Pfeiltasten gedrückt werden.

MainWindow

```
PTWindowsObject MainWindow;
```

MainWindow zeigt auf das Hauptfenster der Anwendung.

nCmdShow

```
int nCmdShow;
```

Ein Integerwert, der angibt, wie das Hauptfenster angezeigt werden soll (normal oder als Symbol). Sehen Sie sich dazu die Windows-Funktion *ShowWindow* an, die die Werte dokumentiert. ObjectWindows verwendet diese Funktion, um das Hauptfenster der Anwendung anzuzeigen.

Elementfunktionen

Konstruktor

```
TApplication (LPSTR AName, HANDLE AnInstance,
HANDLE APrevInstance, LPSTR ACmdLine, int ACmdShow);
```

Erstellt ein *TApplication*-Objekt, initialisiert dessen Datenelemente mit Hilfe der angegebenen Parameter und ruft den *TModule*-Konstruktor auf.

Destruktor

```
~TApplication();
```

Gibt das *TApplication*-Objekt frei. Wenn *MainWindow* noch existiert, wird es gelöscht.

CanClose

```
virtual BOOL CanClose();
```

Liefert *TRUE* zurück, wenn die Anwendung geschlossen werden kann. Per Voreinstellung wird die *CanClose*-Elementfunktion des Hauptfensters aufgerufen und deren Rückgabewert zurückgeliefert. Diese Elementfunktion wird nur selten über-

schrieben; das Schließverhalten kann in der *CanClose*-Elementfunktion des Hauptfensters neu bestimmt werden.

Querverweis: *TWindowsObject::CanClose, TWindowsObject::WMDestroy*

IdleAction `virtual void IdleAction();` protected

Wird während der Botschaftsschleife aufgerufen, wenn die Anwendung »ruht«, das heißt, wenn für eine Anwendung keine Botschaften in der Windows-Botschaftswarteschlange vorhanden sind. Per Vorgabe tut diese Funktion nichts. Sie kann in den abgeleiteteten Klassen überschrieben werden, um irgendeine Aktion auszuführen, während die Anwendung auf Benutzereingaben wartet.

InitApplication `virtual void InitApplication();` protected

Führt eine nur für die erste Instanz der Anwendung erforderliche Initialisierung aus. *InitApplication* unternimmt sonst nichts. Per Voreinstellung führt *InitApplication* keine Aktionen aus. Abgeleitete Klassen können *InitApplication* so überschreiben, daß eine anwendungsspezifische Initialisierung durchgeführt wird.

Querverweis: *TApplication::InitInstance, TApplication::Run*

InitInstance `virtual void InitInstance();` protected

Führt eine für alle Instanzen der Anwendung erforderliche Initialisierung aus. *InitInstance* ruft *InitMainWindow* auf und erstellt und zeigt das Hauptfenster an, indem es *TModule::MakeWindow* und *TWindowsObject::Show* aufruft. Wenn das Hauptfenster nicht erstellt werden konnte, wird das Datenelement *Status* auf *EM_INVALIDMAINWINDOW* gesetzt. Wenn Sie diese Elementfunktion überschreiben, müssen Sie *TApplication::InitInstance* unbedingt explizit aufrufen.

Querverweis: *TApplication::InitMainWindow, TApplication::InitApplication, TApplication::Run, TModule::MakeWindow, TWindowsObject::Show*

InitMainWindow `virtual void InitMainWindow();` protected

InitMainWindow erstellt per Voreinstellung ein generisches *TWindow*-Objekt mit dem Namen der Anwendung als Titel. Überschreiben Sie *InitMainWindow*, um ein Hauptfensterobjekt einer abgeleiteten Klasse zu erstellen und es in *MainWindow* zu speichern. Eine typische Verwendung:

```
virtual void TMyApp::InitMainWindow() {
   MainWindow = new TMyWindow(NULL, "Titel");
}
```

isA `virtual classType isA() const;`

Für Container-Klassen erforderlich. Überschreibt die abstrakte Funktion in der Klasse *Object*. Liefert *applicationClass*, den Klassenbezeichner von *TApplication*, zurück.

MessageLoop `virtual void MessageLoop()` protected

Führt die Botschaftschleife der Anwendung aus, die während der Lebensdauer einer Anwendung abläuft. Frägt Windows nach Botschaften ab und bearbeitet bei Empfang einer Botschaft diese durch den Aufruf von *ProcessAppMsg*. Wenn keine Botschaft vorliegt, ruft *MessageLoop IdleAction* auf, um irgendeine Aktion während der »Ruhezeit« ausführen zu können.

Querverweis: *TApplication::ProcessAppMsg, TApplication::IdleAction*

nameOf `virtual Pchar nameOf() const;`

Für Container-Klassen erforderlich. Überschreibt die abstrakte Funktion in der Klasse *Object*. Liefert "TApplication", den Klassenbezeichnungsstring von *TApplication* zurück.

ProcessAccels `virtual BOOL ProcessAccels(LPMSG PMessage);` protected

Sorgt für eine spezielle Bearbeitung von Tastenkürzelbotschaften. Wenn Ihre Anwendung nicht auf Tastenkürzel antwortet, können Sie die Ausführungsgeschwindigkeit erhöhen, indem Sie die Elementfunktion so überschreiben, daß sie stets *FALSE* zurückliefert.

Querverweis: *TApplication::ProcessAppMsg*

ProcessAppMsg `virtual BOOL ProcessAppMsg(LPMSG PMessage);` protected

Überprüft, ob eine spezielle Botschaftsbearbeitung für nichtmodale Dialoge, Tastenkürzel und MDI-Tastenkürzel erforderlich ist. Ruft *ProcessDlgMsg*, *ProcessMDIAccels* oder *ProcessAccels* auf und liefert *TRUE* zurück, wenn eine dieser speziellen Botschaften angetroffen wird. Wenn Ihre Anwendung keine nicht-modalen Dialoge verwendet, nicht auf Tastenkürzel antwortet und keine MDI-Anwendung ist, können Sie die Ausführungsgeschwindigkeit erhöhen, indem Sie diese Elementfunktion so überschreiben, daß sie stets *FALSE* zurückliefert.

Querverweis: *TApplication::ProcessDlgMsg, TApplication:: ProcessAccels, TApplication::ProcessMDIAccels*

ProcessDlgMsg `virtual BOOL ProcessDlgMsg(LPMSG PMessage);` protected

Sorgt für eine spezielle Bearbeitung von Botschaften nichtmodaler Dialoge und Fenstern und ermöglicht damit Tastatureingaben für Steuerelemente.

Wenn Ihre Anwendung keine nicht-modalen Dialoge oder keine Fenster mit Steuerelementen erstellt, können Sie die Ausführungsgeschwindigkeit erhöhen, indem Sie die Elementfunktion so überschreiben, daß sie stets *FALSE* zurückliefert.

Querverweis: *TApplication::ProcessAppMsg*

ProcessMDIAccels

```
virtual BOOL ProcessMDIAccels(LPMSG PMessage);
```
protected

Sorgt für eine spezielle Bearbeitung von Tastenkürzelbotschaften in MDI-Anwendungen.

Wenn Ihre Anwendung keine MDI-Anwendung ist, können Sie die Ausführungsgeschwindigkeit erhöhen, indem Sie die Elementfunktion so überschreiben, daß sie stets *FALSE* zurückliefert.

Querverweis: *TApplication::ProcessAppMsg*

Run

```
virtual void Run();
```

Initialisiert die Instanz, indem sie *InitApplication* aufruft, wenn noch keine andere Instanz der Anwendung läuft. Danach wird *InitInstance* aufgerufen.

Die Funktion startet die Anwendung durch den Aufruf von *MessageLoop*, wenn die Initialisierung erfolgreich war (d.h. wenn das Datenelement *Status* gleich 0 ist).

Querverweis: *TApplication::InitApplication, TApplication::InitInstance, TApplication::MessageLoop*

SetKBHandler

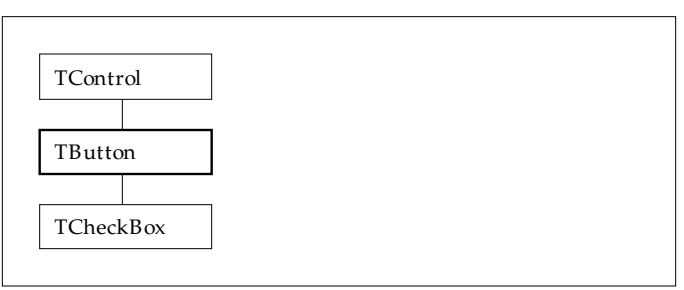
```
void SetKBHandler(PTWindowsObject AWindowsObject);
```

Aktiviert den Bearbeitungsmechanismus für Tastatureingaben (die Übersetzung von Tastatureingaben in Steuerelementauswahlen) für das gegebene Fenster, indem *KBHandlerWnd* auf *AWindowsObject* gesetzt wird. Wird intern von ObjectWindows aufgerufen.

Querverweis: *TWindowsObject::EnableKBHandler, TWindowsObject::WMActivate*

TButton

button.h



TButton ist eine Schnittstellenklasse, die einen Aktionsschalter in Windows repräsentiert. Sie müssen *TButton* verwenden, um ein Schalter-Steuerelement in einem übergeordneten *TWindow*-Fenster zu erstellen. *TButton* vereinfacht auch die Kommunikation zwischen Ihrer Anwendung und den Schaltern eines *TDialog*.

Es gibt zwei Arten von Aktionsschaltern. Ein normaler Schalter wird in einem Rahmen angezeigt. Ein voreingestellter Schalter steht in einem dicken Rahmen und repräsentiert die Standardaktion des Fensters (wenn der Anwender die ⏎-Taste drückt).

In einem Fenster kann es nur einen voreingestellten Schalter geben.

Datenelement

IsDefPB

```
BOOL IsDefPB;
```

IsDefPB zeigt an, ob der Schalter als voreingestellter Schalter angesehen werden soll. Dieses Datenelement wird nur von benutzergezeichneten Schaltern benutzt. Es wird von einem nicht-ressourcenbezogenen *TButton*-Konstruktor in Abhängigkeit vom Stil *BS_DEFPUSHBUTTON* gesetzt.

Elementfunktionen

Konstruktor

```
TButton(PTWindowsObject AParent, int AnId, LPSTR AText,
        int X, int Y, int W, int H, BOOL IsDefault,
        PTModule AModule = NULL);
```

Erstellt ein Schalterobjekt mit dem angegebenen übergeordneten Fenster (*AParent*), mit der Steuerelement-ID (*AnId*), mit dem zugehörigen Text (*AText*), mit der Position (*X*, *Y*), die relativ zum Ursprung des Client-Bereich des übergeordneten Fensters ist, mit der Breite (*W*) und der Höhe (*H*). Der Konstruktor ruft den *TControl*-Konstruktor mit den entsprechenden Parametern auf, und fügt *BS_DEFPUSHBUTTON* zu den für *TButton* (in *Attr.Style*) voreingestellten Stilen hinzu, wenn *isDefault TRUE* ist. Andernfalls wird *BS_PUSHBUTTON* hinzugefügt.

Querverweis: *TControl::TControl*

Konstruktor

```
TButton(PTWindowsObject AParent, int ResourceId,
        PTModule AModule = NULL);
```

Erstellt ein *TButton*-Objekt, das mit einem Schalter eines *TDialog* verknüpft werden soll. Dieser Konstruktor ruft den *TControl*-Konstruktor auf, der über dieselben Parameter verfügt. Danach wird *DisableTransfer* aufgerufen, um den Schalter vom Transfermechanismus auszuschließen, weil Schalter keine Daten haben, die übertragen werden sollen.

Der Parameter *ResourceId* muß einer von Ihnen definierten Schalterressource entsprechen.

Querverweis: *TControl::TControl, TWindowsObject::DisableTransfer*

Konstruktor `TButton(StreamableInit);` protected

Stream-Konstruktor von *TButton*. Ruft den Stream-Konstruktor von *TControl* auf. Er wird aufgerufen, wenn ein *TButton*-Objekt unter Verwendung von Daten aus einem Stream instantiiert wird.

Querverweis: *TControl::TControl*

BMSetStyle
```
virtual void BMSetStyle(RTMessage Msg) =
[WM_FIRST + BM_SETSTYLE];
```

Wenn Windows versucht, den Stil *BS_DEFPUSHBUTTON* zu setzen, stellt diese Elementfunktion für benutzergezeichnete Schalter fest, ob es sich um den voreingestellten Schalter handelt. Ist dies nicht der Fall, ruft diese Elementfunktion lediglich *DefWndProc* auf.

build `static PTStreamable build();`

Ruft den Konstruktor *TButton(StreamableInit)* auf und erstellt ein Objekt vom Typ *TButton*, dessen Datenelemente anschliessend aus einem Stream eingelesen werden.

GetClassName `virtual LPwSTR GetClassName();` protected

Liefert "BUTTON", den Bezeichner, unter dem die Fensterklasse in Windows registriert wird, zurück.

Querverweis: *TWindowsObject::GetWindowClass*

SetupWindow `virtual void SetupWindow();`

Sendet dem übergeordneten Fenster die Botschaft *DM_SETDEFID*, falls es sich bei dem Schalter um den voreingestellten Schalter handelt und wenn er benutzergezeichnet ist.

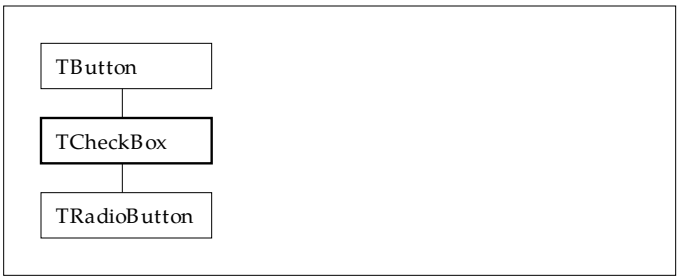
WMGetDlgCode
```
virtual void WMGetDlgCode(RTMessage Msg) =
[WM_FIRST + WM_GETDLGCODE];
```

Beantwortet *WM_GETDLGCODE*-Botschaften der Dialogverwaltung. Für benutzergezeichnete Schalter liefert diese Elementfunktion in der Botschaft die Information zurück, ob es sich um den voreingestellten Schalter handelt.

Für andere Schalter ruft sie lediglich *DefWndProc* auf.

TCheckBox

checkbox.h



TCheckBox ist ein Schnittstellenobjekt, das ein Auswahlfeld in Windows repräsentiert. Sie müssen *TCheckBox* für die Erstellung eines Auswahlfeld in einem übergeordneten *TWindow*-Fenster verwenden. *TCheckBox* vereinfacht die Kommunikation zwischen Ihrer Anwendung und den Auswahlfeldern eines *TDialog*.

Auswahlfelder haben zwei »Zustände«: *ein*- oder *ausgeschaltet*. Zusätzlich gibt es bei manchen Auswahlfeldern einen dritten Status: *deaktiviert*. Die *TCheckBox*-Elementfunktionen kümmern sich in erster Linie um die Bearbeitung des Status von Auswahlfeldern. Ein Auswahlfeld kann auch Teil einer Gruppe (*TGroupBox*) sein, in der die Steuerelemente optisch und funktional gruppiert sind.

Datenelement

Group `PTGroupBox Group`

Group zeigt auf das Steuerelement *TGroupBox*, das das Auswahlfeld logisch mit anderen Auswahl- und Schaltfeldern (*TRadioButton*) gruppiert.

Wenn das Auswahlfeld nicht Teil einer Gruppe ist, ist *Group* gleich *NULL*.

Querverweis: *TGroupBox*, *TRadioButton*

Elementfunktionen

Konstruktor

```
TCheckBox(PTWindowsObject AParent, int AnId,
          LPSTR ATitle, int X, int Y, int W, int H,
          PTGroupBox AGroup, PTModule AModule = NULL);
```

Erstellt ein Auswahlfeld mit dem angegebenen übergeordneten Fenster (*AParent*), mit der Steuerelement-ID (*AnId*), mit dem zugehörigen Text (*ATitle*), mit der Position (*X*, *Y*) relativ zum Ursprung des Client-Bereichs des übergeordneten Fensters, mit der Breite (*W*), der Höhe (*H*) und der zugehörigen Gruppe (*AGroup*).

TCheckBox::TCheckBox setzt das Datenelement *Attr.Style* des Auswahlfeldes auf *WS_CHILD | WS_VISIBLE | WS_TABSTOP | BS_AUTOCHECKBOX* und ruft den *TButton*-Konstruktor auf.

Querverweis: *TButton::TButton*

Konstruktor

```
TCheckBox(PTWindowsObject AParent, int ResourceId,
          PTGroupBox AGroup, PTModule AModule = NULL);
```

Erstellt ein *TCheckBox*-Objekt, das mit einem Auswahlfeld eines *TDialog* verknüpft werden soll. *ResourceId* muß einer Auswahlfeldressource in der Ressourcendatei entsprechen. Die Funktion ruft den *TButton*-Konstruktor auf und setzt *Group* auf *AGroup*. Danach wird durch einen Aufruf von *EnableTransfer* der Transfermechanismus aktiviert.

Querverweis: *TButton::TButton, TWindowsObject::EnableTransfer*

Konstruktor

```
TCheckBox(StreamableInit);
```

protected

Stream-Konstruktor von *TCheckBox*. Ruft den Stream-Konstruktor von *TButton* auf. Er wird aufgerufen, wenn ein *TCheckBox*-Objekt unter Verwendung von Daten aus einem Stream instantiiert wird.

Querverweis: *TButton::TButton*

BNClicked

```
virtual void BNClicked(RTMessage Msg) =
[NF_FIRST + BN_CLICKED];
```

protected

Antwortet automatisch auf Benachrichtigungsbotschaften, die anzeigen, daß ein Auswahlfeld angeklickt wurde.

Wenn *Group* ungleich *NULL* ist, benachrichtigt *BNClicked TGroupBox* durch einen Aufruf der Elementfunktion *SelectionChanged* davon, daß sich der Status geändert hat.

Querverweis: *TGroupBox::SelectionChanged*

build

```
static PTStreamable build();
```

Ruft den Konstruktor *TCheckBox*(*StreamableInit*) auf und erstellt ein Objekt vom Typ *TCheckBox*, dessen Datenelemente anschließ end aus einem Stream eingelesen werden.

Check

```
void Check();
```

Setzt das Auswahlfeld durch einen Aufruf von *SetCheck* in den Status eingeschaltet.

Querverweis: *TCheckBox::SetCheck*

GetCheck

```
WORD GetCheck();
```

Liefert *BN_UNCHECKED* zurück, wenn das Auswahlfeld ausgeschaltet ist, und *BN_CHECKED*, wenn das Auswahlfeld eingeschaltet ist, und *BF_GRAYED*, wenn es deaktiviert ist.

SetCheck `void SetCheck(WORD CheckFlag);`

Bewirkt, daß das Auswahlfeld in den von *CheckFlag* angegebenen Status gesetzt wird. Das Auswahlfeld kann eingeschaltet, ausgeschaltet oder deaktiviert sein, abhängig von *CheckFlag* (*BF_UNCHECKED*, *BF_CHECKED* oder *BF_GRAYED*).

Wenn das Auswahlfeld zu einer Gruppe gehört, informiert *SetCheck* die Gruppe darüber, daß sich die Auswahl geändert hat.

Querverweis: *TGroupBox::SelectionChanged*

read `virtual Pvoid read(Ripstream is);` protected

Ruft *TWindow::read* auf, um das Basisobjekt *TWindow* einzulesen. Dann wird *Group* eingelesen. Liest dann *Group* durch einen Aufruf von *GetSiblingPtr* ein.

Toggle `void Toggle();`

Schaltet den Status des Auswahlfeldes um. Schaltet bei einem Auswahlfeld mit zwei Statusmöglichkeiten zwischen ein- und ausgeschaltet um, bei einem Auswahlfeld mit drei Möglichkeiten zwischen ein-, ausgeschaltet und deaktiviert.

Querverweis: *TCheckBox::SetCheck*

Transfer `virtual WORD Transfer(Pvoid DataPtr, WORD TransferFlag);`

Überträgt den Status eines Auswahlfeldes als Wert vom Typ *WORD* (*BF_CHECKED*, *BF_UNCHECKED* oder *BF_GRAYED*) in oder aus einem Transferpuffer, auf den *DataPtr* zeigt.

Wenn *TransferFlag* auf *TF_GETDATA* gesetzt ist, werden die Status-Daten in den Puffer übertragen. Wenn *TransferFlag* auf *TF_-SETDATA* gesetzt ist, wird das Auswahlfeld auf den Status gesetzt, der im Transferpuffer vorhanden ist. *Transfer* liefert den Umfang der Transferdaten (in Byte) zurück.

Um den Umfang der Daten ohne Transfer ermitteln zu können, setzen Sie *TransferFlag* auf *TF_SIZEDATA*.

Uncheck `void Uncheck();`

Bewirkt, daß das Auswahlfeld durch einen Aufruf von *SetCheck* in den Status ausgeschaltet gesetzt wird.

Querverweis: *TCheckBox::SetCheck*

write `virtual void write(Ropstream os);` protected

Ruft *TWindow::write* auf, um das Basisobjekt *TWindow* in den Stream zu schreiben. Danach wird *Group* an den Stream übergeben. Die Funktion ruft dazu *PutSiblingPtr* auf.

TComboBox

combobox.h

TComboBox ist ein Schnittstellenobjekt, das ein Kombinationsfenster von Windows repräsentiert. Sie müssen *TComboBox* für die Erstellung eines Kombinationsfensters in einem übergeordneten *TWindow*-Fenster verwenden. TComboBox vereinfacht auch die Kommunikation zwischen Ihrer Anwendung und den Kombinationsfenstern eines *TDialog*-Objekts. *TComboBox* erbt den größten Teil seines Verhaltens und seiner Eigenschaften von *TListBox*. Es gibt drei Arten von Kombinationsfenstern: einfache, ausklappende und ausklappende Listen-Kombinationsfenster. Diese drei Fensterarten werden von folgenden Stilkonstanten gesteuert: *CBS_SIMPLE*, *CBS_DROPDOWN* und *CBS_DROPDOWNLIST*. Diese Konstanten werden dem Konstruktor eines *TComboBox*-Objekts übergeben, der Windows mitteilt, welche Art von Kombinationsfenster erstellt werden soll.

Datenelement

TextLen

```
WORD TextLen;
```

TextLen-1 ist die maximale Länge des Zeichenpuffers im Editierbereich des Kombinationsfenster. *TextLen* wird vom *TComboBox*-Konstruktor gesetzt.

Elementfunktionen

Konstruktor

```
TComboBox(PTWindowsObject AParent, int AnId, int X,
          int Y, int W, int H, DWORD AStyle,
          WORD ATextLen, PTModule AModule = NULL);
```

Erstellt ein Kombinationsfenster mit dem angegebenen übergeordneten Fenster (*AParent*), der Steuerelement-ID (*AnId*), der Position (*X*, *Y*) relativ zum Ursprung des Client-Bereichs des übergeordneten Fensters, der Breite (*W*) und der Höhe (*H*), im Stil (*AStyle*) und mit der Textlänge (*ATextLen*).

Ruft den *TListBox*-Konstruktor auf, der über ähnliche Parameter verfügt. Danach wird *Attr.Style* folgendermaßen gesetzt:

```
Attr.Style = WS_CHILD | WS_VISIBLE | WS_GROUP |
             WS_TABSTOP | CBS_SORT | CBS_AUTOHSCROLL |
             WS_VSCROLL | AStyle;
```

Eine der folgenden Stilkonstanten für Kombinationsfenster muß sich unter den in *AStyle* gesetzten Konstanten befinden: *CBS_SIMPLE*, *CBS_DROPDOWN* , *CBS_DROPDOWNLIST* , *CBS_OWNERDRAWFIXED* oder *CBS_OWNERDRAWVARIABLE* .

Querverweis: *TComboBox::TextLen*, *TListBox::TListBox*

Konstruktor
```
TComboBox(PTWindowsObject AParent, int ResourceId,
          WORD ATextLen, PTModule AModule = NULL);
```

Erstellt ein *TComboBox*-Objekt, das mit einem Kombinationsfenster eines *TDialog*-Objekts verknüpft werden soll, und ruft den *TListBox*-Konstruktor auf, der über ähnliche Parameter verfügt. Danach wird das *TextLen*-Element auf *ATextLen* gesetzt. Der Parameter *ResourceId* muß einer von Ihnen definierten Kombinationsfensterressource entsprechen.

Querverweis: *TListBox::TListBox*

Konstruktor
```
TComboBox(StreamableInit);
```
protected

Stream-Konstruktor von *TComboBox*. Ruft den Stream-Konstruktor von *TListBox* auf. Er wird aufgerufen, wenn ein *TComboBox*-Objekt unter Verwendung von Daten aus einem Stream instantiiert wird.

Querverweis: *TListBox::TListBox*

build
```
static PTStreamable build();
```

Ruft den Konstruktor *TComboBox(StreamableInit)* auf und erstellt ein Objekt vom Typ *TComboBox*, dessen Datenelemente anschließend aus einem Stream eingelesen werden.

Clear
```
void Clear();
```

Löscht den Text des zugehörigen Editierfeldes.

GetClassName
```
virtual LPSTR GetClassName();
```
protected

Liefert "COMBOBOX", den Bezeichner, unter dem die Fensterklasse in Windows registriert wird, zurück.

Querverweis: *TWindowsObject::GetWindowClass*

GetEditSel
```
int GetEditSel(Rint StartPos, Rint EndPos);
```

Liefert in den angebenen Referenzparametern die Anfangs- und Endpositionen des im zugehörigen Editierfeld markierten Textes. Liefert *CB_ERROR*, wenn das Kombinationsfenster kein Editierfeld hat.

GetMsgID
```
virtual WORD GetMsgIDd(WORD AnId);
```
protected

Liefert den Bezeichner einer Windows-Kombinationsfensterbotschaft, die mit dem angegebenen ObjectWindows-Botschaftsbezeichner verknüpft ist.

GetText
```
int GetText(LPSTR AString, int MaxChars);
```
Kopiert in den angegebenen String den Text des zugehörigen Editierfeldes (maximal bis zur Länge *MaxChars*) und liefert die Anzahl der kopierten Zeichen.

GetTextLen
```
int GetTextLen();
```
Liefert die Länge des Textes im zugehörigen Editierfeld.

Clear
```
void Clear();
```
Löscht den Text des zugehörigen Editierfeldes.

HideList
```
void HideList();
```
Verbirgt die Liste in einem ausklappenden Kombinationsfenster oder einem ausklappenden Listen-Kombinationsfenster.

Querverweis: *TComboBox::ShowList*

nameOf
```
virtual Pchar nameOf() const;
```
Für Container-Klassen erforderlich. Überschreibt die abstrakte Funktion in der Klasse *Object*. Liefert "TComboBox", den Klassenbezeichnungsstring von *TComboBox*, zurück.

read
```
virtual Pvoid read(Ripstream is);
```
protected

Ruft *TListBox::read* auf, um das Basisobjekt *TListBox* einzulesen. Danach wird *TextLen* eingelesen.

SetEditSel
```
int SetEditSel(int StartPos, int EndPos);
```
Markiert die Zeichen, die zwischen *StartPos* und *EndPos* im Editierfeld des Kombinationsfensters stehen.

Liefert *CB_ERROR*, wenn das Kombinationsfenster kein Editierfeld hat.

SetText
```
void SetText(LPSTR AString);
```
Wählt den ersten String in der zugehörigen Liste aus, der mit dem angegebenen *AString* beginnt.

Kann keine Entsprechung gefunden werden, wird der Inhalt des zugehörigen Editierfeldes auf den angegebenen String gesetzt (und markiert).

SetupWindow
```
virtual void SetupWindow();
```
protected

Baut das Kombinationsfenster auf und limitiert dabei die Textlänge in seinem Editierfeld auf *TextLen*-1.

ShowList
```
void ShowList();
```
Zeigt die Liste eines ausklappenden Kombinationsfensters oder eines ausklappenden Listen-Kombinationsfensters an.

Querverweis: *TComboBox::HideList*

Transfer `virtual WORD Transfer(Pvoid DataPtr, WORD TransferFlag);`

Überträgt die Einträge und die Auswahl des Kombinationsfensters in einen oder aus einem Transferpuffer, je nachdem ob *TransferFlag* auf *TF_SETDATA* oder *TF_GETDATA* gesetzt ist. *TransferBuffer* soll auf ein *PTComboBoxData*-Objekt zeigen, das als Transferpuffer für ein Kombinationsfenster dient.

Transfer liefert die Größe von *PTComboBoxData* (den Zeiger, nicht die Struktur) zurück. Wenn sie ermittelt werden soll, ohne daß ein Transfer stattfindet, setzen Sie *TransferFlag* auf *TF_SIZEDATA*.

Hinweis: Sie müssen in Ihrem Transferpuffer einen Zeiger auf diese Strukturen verwenden. Sie können keine Kopien dieser Strukturen in Ihren Transferpuffer einbringen und Sie können diese Strukturen nicht als Transferpuffer verwenden.

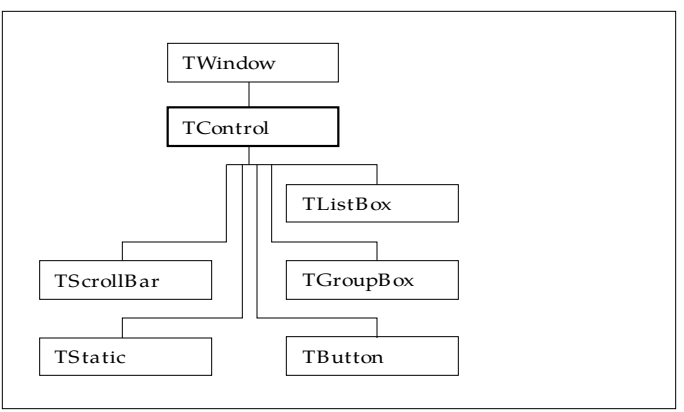
Querverweis: *ListBox::Transfer*

write `virtual void write(Ropstream os);` protected

Ruft *TListBox::write* auf, um zuerst das Basisobjekt *TListBox* und danach *Textlen* in den Stream zu schreiben.

TControl

control.h



TControl umfaßt abgeleitete Steuerelementklassen, wie *TScrollBar* und *TButton*. Steuerelementobjekte abgeleiteter Klassen werden zur Repräsentation von Steuerelementen in Windows verwendet. Für die Erstellung eines Steuerelementes in einem übergeordneten *TWindow*-Fenster muß ein Steuerelementobjekt verwendet werden. Ein Steuerelementobjekt kann die Kommunikation zwischen Ihrer Anwendung und den Steuerelementen eines *TDialog*-Objekts vereinfachen.

Element-funktionen

Konstruktor

```
TControl(PTWindowsObject AParent, int AnId,
         LPSTR ATitle, int X, int Y, int W,
         int H, PTModule AModule = NULL);
```

Ruft den *TWindow*-Konstruktor auf und übergibt ihm *AParent* (das übergeordnete Fenster), *ATitle* (den Titeltext) und *AModule*. Setzt *Attr* mit Hilfe des angegebenen Fensterbezeichners (*AnId*), der Position (*X*, *Y*) relativ zum Ursprung des Client-Bereichs des übergeordneten Fensters, der Breite (*W*) und der Höhe (*H*). Setzt *Attr.Style* auf *WS_CHILD* | *WS_VISIBLE* | *WS_-GROUP* | *WS_TABSTOP*.

Querverweis: *TWindow::TWindow*

Konstruktor

```
TControl(PTWindowsObject AParent, int ResourceId,
         PTModule AModule = NULL);
```

Erstellt ein Steuerelement, das mit einem *TDialog*-Objekt verknüpft ist. Dieser Konstruktor ruft den *TWindow*-Konstruktor auf und aktiviert dann den Transfermechanismus durch einen Aufruf von *EnableTransfer*.

Der Parameter *ResourceId* muß einer von Ihnen definierten Steuerelementressource entsprechen.

Querverweis: *TWindow::TWindow, TWindowsObject::Enable-Transfer*

Konstruktor

```
TControl(StreamableInit);
```

protected

Stream-Konstruktor von *TControl*. Ruft den Stream-Konstruktor von *TWindow* auf. Er wird dann aufgerufen, wenn ein *TControl*-Objekt unter Verwendung von Daten aus einem Stream instantiiert wird.

Querverweis: *TWindow::TWindow*

GetId

```
virtual int GetId();
```

Liefert den Fensterbezeichner *Attr.Id*.

ODADraw-Entire

```
virtual void ODADrawEntire(DRAWITEMSTRUCT
                           _FAR & DrawInfo);
```

protected

Antwortet auf eine Nachricht, die an ein benutzergezeichnetes Steuerelement geschickt wird, wenn es gezeichnet werden muß.

Per Voreinstellung wird *Parent->DrawItem* aufgerufen. Kann von dem benutzergezeichneten Steuerelement überschrieben werden, um festzulegen, wie es gezeichnet werden soll.

Querverweis: *TWindowsObject::DrawItem*, *TControl::WMDraw-Item*

ODAFocus
```
virtual void ODAFocus(DRAWITEMSTRUCT
                      _FAR & DrawInfo);
```
protected

Antwortet auf eine Nachricht, die an ein benutzergezeichnetes Steuerelement geschickt wird, wenn der Fokus an ein anderes Steuerelement übergeben wurde. Per Voreinstellung wird *Parent->DrawItem* aufgerufen. Kann von dem benutzergezeichneten Steuerelement überschrieben werden, um festzulegen, wie es gezeichnet werden soll, wenn es den Fokus erhält oder abgibt.

Querverweis: *TWindowsObject::DrawItem* , *TWindowsObject::-WMDrawItem*

ODASelect
```
virtual void ODASelect(DRAWITEMSTRUCT _FAR & DrawInfo);
```
protected

Antwortet auf eine Nachricht, die an ein benutzergezeichnetes Steuerelement geschickt wird, wenn sich der Status des Steuerelements ändert. Per Voreinstellung wird *Parent->DrawItem* aufgerufen. Kann vom benutzergezeichneten Steuerelement überschrieben werden, um festzulegen, wie es gezeichnet werden soll, wenn sich der Auswahl-Status ändert.

Querverweis: *TWindowsObject::DrawItem* , *TWindowsObject::-WMDrawItem*

WMDrawItem
```
virtual void WMDrawItem(RTMessage Msg) =
[WM_FIRST + WM_DRAWITEM];
```

Antwortet auf eine *WM_DRAWITEM* -Botschaft, die an ein benutzergezeichnetes Steuerelement geschickt wird, wenn das Steuerelement gezeichnet werden soll. *TControl::WMDrawItem* ruft *ODADrawEntire* auf, wenn das gesamte Steuerelement gezeichnet werden soll.

Wenn sich der Auswahl-Status des Steuerelements geändert hat, wird *ODASelect* aufgerufen. Wenn der Fokus zu oder von einem anderen Steuerelement übergeben wurde, wird *ODA-Focus* aufgerufen.

Querverweis: *TControl::ODADrawEntire*, *TControl::ODASelect*, *TControl::ODAFocus*, *TWindowsObjekt::WMDrawItem*

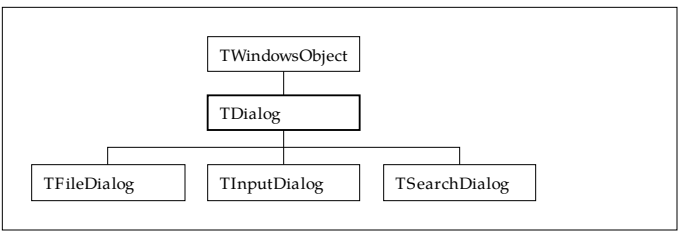
WMPaint
```
virtual void WMPaint(RTMessage Msg) =
[WM_FIRST + WM_PAINT];
```

Wenn das Steuerelement eine vordefinierte Windows-Klasse hat, wird *DefWndProc* für Windows-unterstütztes Zeichnen aufgerufen; andernfalls erfolgt ein *TWindow::WMPaint* -Aufruf.

Querverweis: *TWindow::WMPaint*

TDialog

dialog.h



TDialog-Objekte repräsentieren modale und nicht-modale Dialoge. Zu einem *TDialog*-Objekt gehört eine entsprechende Ressourcendefinition, welche die Position und das Aussehen seiner Steuerelemente beschreibt. Der Bezeichner dieser Ressourcendefinition wird im *TDialog*-Konstruktor angegeben.

Ein *TDialog*-Objekt wird durch den Aufruf der Elementfunktionen *Execute* oder *Create* mit einem modalen oder nicht-modalen Schnittstellenelement verknüpft.

Diese Elementfunktionen werden jedoch normalerweise nicht direkt in einer ObjectWindows-Anwendung aufgerufen. Stattdessen erfolgt ein Aufruf der *TModule*-Elementfunktionen *ExecDialog* und *MakeWindow*. Diese Funktionen überprüfen vor der Verknüpfung ein *TDialog*-Objekt auf seine Gültigkeit. Beachten Sie, daß ein modaler Dialog Operationen in seinem übergeordneten Fenster ausschließt, solange er geöffnet ist.

Datenelemente

Attr

```
TDialogAttr Attr;
```

Attr speichert die Erstellungsattribute des Dialogs. Die Definition von *TDialogAttr* ist in Kapitel 24 »Weitere Komponenten von ObjectWindows« im vorliegenden Handbuch erläutert.

IsModal

```
BOOL IsModal;
```

IsModal ist *TRUE*, wenn es sich um einen modalen Dialog handelt, und *FALSE*, wenn der Dialog nicht-modal ist.

Elementfunktionen

Konstruktor

```
TDialog(PTWindowsObject AParent, LPSTR AName,
        PTModule AModule = NULL);
```

Ruft den *TWindowsObject*-Konstruktor auf und übergibt diesem *AParent* und *AModule*. *DisableAutoCreate* wird aufgerufen, damit *TDialog* nicht automatisch mit seinem übergeordneten Fenster erstellt und angezeigt wird. *Title* wird mit -1 initialisiert. Weist *Attr.Name* eine Kopie von *AName* zu. *AName* ist ein String-Ressourcenbezeichner, den Sie in der Ressourcen-

Definitionsdatei definieren müssen. Die Funktion initialisiert *Attr.Param* mit 0 und setzt *IsModal* auf *FALSE*.

Querverweis: *TWindowsObject::TWindowsObject , TWindows-Object::DisableAutoCreate*

Konstruktor

```
TDialog(PTWindowsObject AParent, int ResourceId,
        PTModule AModule = NULL);
```

Ruft einen *TWindowsObject*-Konstruktor auf und übergibt diesem *AParent* und *AModule*. Ruft *DisableAutoCreate* auf, damit *TDialog* nicht automatisch mit seinem übergeordneten Fenster erstellt und angezeigt wird. *Title* wird mit -1 initialisiert. Setzt *Attr.Name* mit Hilfe des Ressourcenbezeichners (einem Integerwert) des Dialogs. Dieser muß einer Dialog-Ressourcendefinition in der Ressourcendatei entsprechen. *Attr.Param* wird auf 0 und *IsModal* auf *FALSE* gestzt.

Querverweis: *TWindowsObject::TWindowsObject , TWindows-Object::DisableAutoCreate*

Konstruktor

```
TDialog(StreamableInit);
```
protected

Stream-Konstruktor von *TDialog*. Ruft den Stream-Konstruktor von *TWindowsObject* auf. Er wird aufgerufen, wenn ein *TDialog*-Objekt unter Verwendung von Daten aus einem Stream instantiiert wird.

Querverweis: *TWindowsObject::TWindowsObject*

Destruktor

```
virtual ~TDialog();
```

Wenn es sich beim Ressourcenbezeichner in *Attr.Name* um einen String und nicht um einem Integerwert handelt, wird der Speicherplatz freigegeben, in dem der Name des *TDialog*-Objekts (*Attr.Name*) gespeichert ist.

Querverweis: *TWindowsObject::~TWindowsObject*

build

```
static PTStreamable build();
```

Ruft den Konstruktor *TDialog*(*StreamableInit*) auf und erstellt ein Objekt vom Typ *TDialog*, dessen Datenelemente anschließend aus einem Stream eingelesen werden.

Cancel

```
virtual void Cancel(RTMessage Msg) =
[ID_FIRST + IDCANCEL];
```
protected

Antwortet automatisch auf das Anklicken des *Cancel*-Schalters (Abbrechen) in einem Dialog. Ruft *CloseWindow*(*Int*) auf und übergibt *IDCANCEL*.

Querverweis: *TDialog::CloseWindow*

CloseWindow `virtual void CloseWindow(int ARetValue);`

Schließt unter bestimmten Bedingungen das Fenster. Wenn this ein nicht-modaler Dialog ist, wird *TWindowsObject::CloseWindow* aufgerufen. Wenn this ein modaler Dialog ist, wird *CanClose* aufgerufen. Falls *CanClose TRUE* zurückliefert, erfolgt ein Aufruf von *TransferData*, um die Dialog-Daten zu übertragen. Schließlich wird *ShutDownWindow* aufgerufen und *ARetValue* übergeben.

Querverweis: *TWindowsObject::CloseWindow, TDialog::CanClose, TDialog::TransferData, TDialog::ShutDownWindow*

CloseWindow `virtual void CloseWindow();`

Schließt unter bestimmten Bedingungen den Dialog. Diese Funktion ruft *TWindowsObject::CloseWindow* auf, wenn this ein nicht-modaler Dialog ist. Ist this ein modaler Dialog, ruft diese Funktion *CloseWindow*(*IDCANCEL*) auf.

Querverweis: *TWindowsObject::CloseWindow* (*int ARetValue*)

Create `virtual BOOL Create();`

Erstellt ein modales Dialog-Schnittstellenelement, das mit dem *TDialog*-Objekt verknüpft ist. Die Verknüpfung findet nicht statt, wenn *Status* ungleich 0 ist.

Ruft *DisableCreate* auf, um eine automatische Erstellung von untergeordneten Fenstern zu verhindern. Durch den Aufruf von *EnableKBHandler* wird der Mechanismus zur Bearbeitung von Tastatureingaben aktiviert.

Die Funktion registriert alle untergeordneten Fenster des Dialogs, um die Verwendung von selbstdefinierten Steuerelementen zu ermöglichen, und ruft die Windows-Funktion *CreateDialogParam* für die Erstellung des Dialogs auf. Bei erfolgreicher Bearbeitung liefert *Create TRUE* zurück; andernfalls wird *Status* auf *EM_INVALIDWINDOW* gesetzt und *Create* liefert *FALSE* zurück.

Wichtig: Die Verwendung von *TModule::MakeWindow* ist die sicherere Methode zur Erstellung eines nicht-modalen Dialogs.

Querverweis: *TDialog::Execute, TModule::MakeWindow, TWindowsObject::DisableAutoCreate, TWindowsObject::EnableKBHandler*

Destroy `virtual void Destroy(int ARetValue);`

Gibt das zu *TDialog* gehörige Schnittstellenelement frei. Ist this ein nicht-modaler Dialog, wird *TWindowsObject::Destroy* aufgerufen. Ist this ein modaler Dialog, wird *EnableAutoCreate* für alle untergeordneten Fenster aufgerufen. Danach erfolgt ein Aufruf der Windows-Funktion *EndDialog* und *ARetValue* wird

übergeben, damit dieser Wert als Ergebnis der Dialog-Durchführung zurückgeliefert wird.

Querverweis: *TWindowsObject::Destroy*, *TWindowsObject::EnableAutoCreate*

Destroy `virtual void Destroy();`

Gibt das zu *TDialog* gehörige Schnittstellenelement frei. Wenn this ein nicht-modaler Dialog ist, dann wird *TWindowsObject::Destroy* aufgerufen. Ist this ein modaler Dialog, wird *Destroy* (*IDCANCEL*) aufgerufen.

Querverweis: *TWindowsObject::Destroy* (*int ARetValue*)

Execute `virtual int Execute();`

Erstellt ein modales Dialog-Schnittstellenelement aus, das mit dem *TDialog*-Objekt verknüpft ist, und führt den Dialog aus. Die Verknüpfung findet nicht statt, wenn das *Status*-Element ungleich 0 ist. Wenn eine erfolgreiche Verknüpfung möglich war, kehrt *Execute* erst zurück, wenn *TDialog* geschlossen wird.

Ruft *DisableAutoCreate* auf, um eine automatische Erstellung aller untergeordneten Fenstern zu verhindern. Durch den Aufruf von *EnableKBHandler* wird der Tastatur-Bearbeitungsmechanismus aktiviert. Die Funktion registriert alle untergeordneten Fenster des Dialogs, um die Verwendung von selbstdefinierten Steuerelementen zu unterstützen, und ruft die Windows-Funktion *DialogBoxParam* zur Durchführung des Dialogs auf.

Wichtig: Die Verwendung von *TModule::ExecDialog* ist die sicherere Methode zur Erstellung eines modalen Dialogs.

Querverweis: *TModule::ExecDialog*, *TWindowsObject::DisableAutoCreate*, *TWindowsObject::EnableKBHandler*

GetClassName `virtual LPSTR GetClassName();` protected

Liefert den Namen der vordefinierten Windows-Fensterklasse zurück, die für einen modalen Dialog verwendet werden muß. Liefert bei nicht-modalen Dialogen den Namen "OWLDialog" der ObjectWindows-Standardklasse zurück.

GetItemHandle `HWND GetItemHandle(int DlgItemID);`

Liefert das Fenster-Handle des Dialog-Steuerelements zurück, das von der ID *DlgItemID* bezeichnet wird.

GetWindowClass `virtual void GetWindowClass(WNDCLASS _FAR & AWndClass);` protected

Setzt *AWndClass* auf die voreingestellten Registrierungsattribute für ein *TDialog*-Objekt. Per Voreinstellung verfügt das *TDialog*-Objekt über ein Standard-Anwendungssymbol und einen Pfeil-Mauszeiger. Um diese oder andere voreingestellte

Registrierungsattribute zu ändern, überschreiben Sie diese Elementfunktion (und ebenfalls die Funktion *GetClassName*) in der abgeleiteten *TDialog*-Klasse.

Bevor Sie die gesetzten Voreinstellungen ändern, müssen Sie *TDialog::GetWindowClass* in der neuen *GetWindowClass*-Funktion aufrufen.

Querverweis: *TWindowsObject::GetClassName*

isA

```
virtual classType isA() const;
```

Für Container-Klassen erforderlich. Überschreibt die abstrakte Funktion der Klasse *Object*. Liefert *dialogClass*, den Objekt-Bezeichner von *TDialog*, zurück.

nameOf

```
virtual Pchar nameOf() const;
```

Für Container-Klassen erforderlich. Überschreibt die abstrakte Funktion in der Klasse *Object*. Liefert "TDialog", den Klassenbezeichnungsstring von *TDialog*, zurück.

Ok

```
virtual void Ok(RTMessage Msg) =
[ID_FIRST + IDOK];                                    protected
```

Beantwortet das Anklicken des *OK*-Schalters (mit dem Bezeichner *IDOK*) in einem Dialog. Ruft *CloseWindow* auf und übergibt *IDOK*.

Querverweis: *TWindow::CloseWindow*

read

```
virtual Pvoid read(Ripstream is);                     protected
```

Ruft *TWindowsObject::read* auf, um das Basisobjekt *TWindowsObject* einzulesen. Danach werden *Attr.Name* und *IsModal* gelesen.

SendDlgItemMsg

```
DWORD SendDlgItemMsg(int DlgItemID, WORD AMsg,
                     WORD WParam, DWORD LParam);
```

Sendet eine Windows-Steuerelementbotschaft, die von *AMsg* angegeben wird, zu dem von der ID *DlgItemID* bestimmten Dialog-Steuerelement. *WParam* und *LParam* werden zu Parametern der Steuerelementbotschaft.

SendDlgItemMsg liefert den vom Steuerelement zurückgegebenen Wert zurück bzw. den Wert 0, wenn die Steuerelement-ID nicht zulässig ist.

SetCaption

```
void SetCaption(LPSTR ATitle);
```

Wenn *ATitle* nicht -1 ist, wird *TWindowsObject::SetCaption* aufgerufen.

Querverweis: *TDialog::SetupWindow, TWindowsObject::SetCaption*

SetupWindow `virtual void SetupWindow();` protected

Baut mit einem Aufruf von *SetCaption* (setzt *Title*) und *TWindowsObject::SetupWindow* den Dialog auf.

Querverweis: *TDialog::SetCaption, TWindowsObject:.SetupWindow*

ShutDownWindow `virtual void ShutDownWindow(int ARetValue);`

Schließt den Dialog. Ist this ein nicht-modaler Dialog, wird *TWindowsObject::ShutDownWindow* aufgerufen, ist es ein modaler Dialog, wird *Destroy* aufgerufen und *ARetValue* übergeben.

Querverweis: *TWindowsObject::ShutDownWindow, TDialog::-Destroy*

ShutDownWindow `virtual void ShutDownWindow();`

Schließt den Dialog. Wenn this ein nicht-modaler Dialog ist, wird *TWindowsObject::ShutDownWindow* aufgerufen. Ist this ein modaler Dialog, wird *Destroy* aufgerufen und *IDCANCEL* übergeben.

Querverweis: *TWindowsObject::ShutDownWindow, TDialog::-Destroy*

WMClose `virtual void WMClose(RTMessage Msg) = [WM_FIRST + WM_CLOSE];` protected

Die Funktion antwortet auf eine eingehende *WM_CLOSE*-Botschaft, indem sie das Fenster schließt.

Querverweis: *TDialog::ShutDownWindow*

WMInitDialog `virtual void WMInitDialog(RTMessage Msg) = [WM_FIRST + WM_INITDIALOG];` protected

WMInitDialog wird vor der Anzeige des Dialogs automatisch aufgerufen. Die Funktion ruft *SetupWindow* auf, um den erforderlichen Aufbau des Dialogs und seiner Steuerelemente durchzuführen.

Querverweis: *TWindowsObject::SetupWindow*

WMQueryEndSession `virtual void WMQueryEndSession(RTMessage Msg) = [WM_FIRST + WM_QUERYENDSESSION];` protected

Die Funktion antwortet auf den Versuch von Windows, die Sitzung zu beenden. Wenn this das Hauptfenster ist, ruft sie *TApplication->CanClose* auf, andernfalls *this->CanClose*. Liefert 0 zurück (*setzt Msg.Result*), wenn der Aufruf von *CanClose TRUE* zurückgibt und damit anzeigt, daß die Sitzung beendet werden kann; andernfalls wird 1 zurückgeliefert (weil ein Dialog sich umgekehrt verhalten muß wie ein normales Fenster).

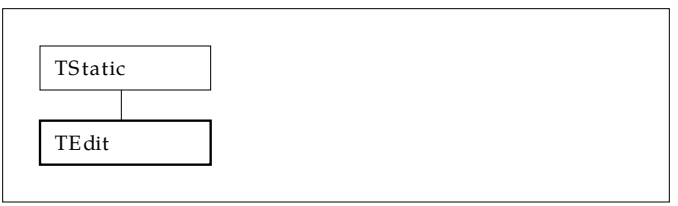
Wichtig: WM_QUERYENDSESSION von Windows verhält sich umgekehrt.

write `virtual void write(Ropstream os);` protected

Ruft *TWindowsObject::write* auf, um das Basisobjekt *TWindowsObject* in den Stream zu schreiben. Danach werden *Attr.Name* und *isModal* in den Stream geschrieben.

TEdit

edit.h



TEdit ist ein Schnittstellenobjekt, das ein Editierfeld in Windows repräsentiert. Für die Erstellung eines Editierfeldes in einem übergeordneten *TWindow*-Objekt muß ein *TEdit*-Objekt verwendet werden. *TEdit* vereinfacht auch die Kommunikation zwischen Ihrer Anwendung und den Editierfeldern eines *TDialog*-Objekts.

Es gibt zwei Arten von Editierfeldern: einzeilige und mehrzeilige. Mehrzeilige Editierfelder ermöglichen das Editieren mehrerer Textzeilen. Beachten Sie, daß die Position des ersten Zeichens in einem Editierfeld 0 ist. In einem mehrzeiligen Editierfeld werden die Positionsnummern kontinuierlich von Zeile zu Zeile weitergezählt. Zeilenumbrüche gelten als zwei Zeichen.

Die meisten Elementfunktionen von *TEdit* dienen zur Bearbeitung des Textes in Editierfeldern. *TEdit* verfügt zudem über einige Botschaftsantwortfunktionen, die die Auswahl von *Cut*, *Copy* und *Paste* in dem Menü beantworten, das zu dem dem Editierfenster übergeordneten Fenster gehört. Die beiden wichtigen Elementfunktionen *GetText* und *SetText* sind Erbgut von der *TEdit*-Basisklasse *TStatic*.

Elementfunktionen

Konstruktor

```
TEdit(PTWindowsObject AParent, int AnId, LPSTR AText,
      int X, int Y, int W, int H, WORD ATextLen,
      BOOL Multiline, PTModule AModule = NULL);
```

Erstellt ein Editierfeld-Objekt mit einem übergeordneten Fenster (*AParent*). Setzt die Erstellungsattribute des Editierfeldes und füllt seine *Attr*-Datenelemente mit der angegebenen Steuerelement-ID (*AnId*), mit der Position (*X*, *Y*) relativ zum Ursprung des Client-Bereichs des übergeordneten Fensters, mit der Breite (*W*) und der Höhe (*H*) aus. Ist die Länge des Textpuf-

fers (*ATextLen*) gleich Null oder 1, besteht keine Beschränkung für die Anzahl der Zeichen, die eingegeben werden können. Andernfalls können *ATextLen* -1 Zeichen eingegeben werden. Per Voreinstellung wird der Text (*AText*), mit dem das Editierfeld initialisiert wird, linksbündig ausgerichtet und das Editierfeld mit einem Rahmen versehen. Mehrzeilige Editierfelder verfügen über vertikale und horizontale Bildlaufleisten.

Konstruktor

```
TEdit(PTWindowsObject AParent, int ResourceId, WORD
      ATextLen, PTModule AModule = NULL);
```

Erstellt ein *TEdit*-Objekt, das mit einem Editierfeld eines *TDialog* verknüpft werden soll, und ruft den *TStatic*-Konstruktor mit identischen Parametern auf.

Der Parameter *ResourceId* muß einer von Ihnen definierten Editierressource entsprechen. Die Funktion schaltet durch den Aufruf von *EnableTransfer* den Transfermechanismus ein.

Querverweis: *TStatic::TStatic*

Konstruktor

```
TEdit(StreamableInit);
```

protected

Stream-Konstruktor von *TEdit*. Ruft den Stream-Konstruktor von *TStatic* auf. Er wird aufgerufen, wenn ein *TEdit*-Objekt unter Verwendung von Daten aus einem Stream instantiiert wird.

Querverweis: *TStatic::TStatic*

build

```
static PTStreamable build();
```

Ruft den Konstruktor *TEdit(StreamableInit)* auf und erstellt ein Objekt vom Typ *TEdit*, dessen Datenelemente anschließend aus einem Stream eingelesen werden.

CanUndo

```
BOOL CanUndo();
```

Liefert *TRUE* zurück, wenn die letzte Änderung rückgängig gemacht werden kann.

Querverweis: *TEdit::Undo*

ClearModify

```
void ClearModify();
```

Setzt das Änderungs-Flag des Editierfeldes neu. Dadurch liefert *IsModified FALSE* zurück. Das Flag wird gesetzt, wenn der Text geändert wird.

Querverweis: *TEdit::IsModified*

CMEditClear

```
virtual void CMEditClear(RTMessage Msg) =
[CM_FIRST + CM_EDITCLEAR];
```

protected

Antwortet durch den Aufruf der Elementfunktion *Clear* auf eine Menüauswahl mit der Menü-ID *CM_EDITCLEAR*.

Querverweis: *TStatic::Clear*

CMEditCopy
```
virtual void CMEditCopy(RTMessage Msg) =
[CM_FIRST + CM_EDITCOPY];                    protected
```

Antwortet durch den Aufruf der Elementfunktion *Copy* auf eine Menüauswahl mit der Menü-ID *CM_EDITCOPY*.

Querverweis: *TEdit::Copy*

CMEditCut
```
virtual void CMEditCut(RTMessage Msg) =
[CM_FIRST + CM_EDITCUT];                     protected
```

Antwortet durch den Aufruf der Elementfunktion *Cut* auf eine Menüauswahl mit der Menü-ID *CM_EDITCUT*.

Querverweis: *TEdit::Cut*

CMEditDelete
```
virtual void CMEditDelete(RTMessage Msg) =
[CM_FIRST + CM_EDITDELETE];                  protected
```

Antwortet durch den Aufruf der Elementfunktion *DeleteSelection* auf eine Menüauswahl mit der Menü-ID *CM_EDITDELETE*.

Querverweis: *TEdit::DeleteSelection*

CMEditPaste
```
virtual void CMEditPaste(RTMessage Msg) =
[CM_FIRST + CM_EDITPASTE];                   protected
```

Antwortet durch den Aufruf der Elementfunktion *Paste* auf eine Menüauswahl mit der Menü-ID *CM_EDITPASTE*.

Querverweis: *TEdit::Paste*

CMEditUndo
```
virtual void CMEditUndo(RTMessage Msg) =
[CM_FIRST + CM_EDITUNDO];                    protected
```

Antwortet durch den Aufruf der Elementfunktion *Undo* auf eine Menüauswahl mit der Menü-ID *CM_EDITUNDO*.

Querverweis: *TEdit::Undo*

Copy
```
void Copy();
```

Kopiert den aktuell ausgewählten Text in die Zwischenablage.

Querverweis: *TEdit::CMEditCopy*

Cut
```
void Cut();
```

Löscht den aktuell ausgewählten Text und kopiert ihn in die Zwischenablage.

Querverweis: *TEdit::CMEditCut*

DeleteLine
```
BOOL DeleteLine(int LineNumber);
```

Löscht in einem mehrzeiligen Editierfeld den Text in der mit *LineNumber* angegebenen Zeile. Bei Übergabe von -1 wird die aktuelle Zeile gelöscht. *DeleteLine* löscht nicht den Zeilenumbruch und tastet die anderen Zeilen nicht an. Liefert bei

erfolgreicher Durchführung *TRUE* zurück. Liefert *FALSE*, wenn *LineNumber* nicht -1 ist und außerhalb des Bereichs liegt oder wenn ein Fehler aufgetreten ist.

DeleteSelection
```
BOOL DeleteSelection();
```

Löscht den aktuell ausgewählten Text und liefert *FALSE* zurück, wenn kein Text ausgewählt ist.

Querverweis: *TEdit::CMEditDelete*

DeleteSubText
```
BOOL DeleteSubText(int StartPos, int EndPos);
```

Löscht den Text zwischen den Anfangs- und Endpositionen, die in *StartPos* und *EndPos* angegeben werden. *DeleteSubText* liefert bei erfolgreicher Durchführung *TRUE* zurück.

ENErrSpace
```
virtual void ENErrSpace(RTMessage Msg) =
[NF_FIRST + EN_ERRSPACE];
```
protected

Antwortet auf eine Fehlerbenachrichtigungsbotschaft, indem es einen Signalton ausgibt. Diese Botschaft wird gesendet, wenn das Editierfeld erfolglos versucht hat, weiteren Speicherplatz zu reservieren.

GetClassName
```
virtual LPSTR GetClassName();
```
protected

Liefert "EDIT", den Bezeichner, unter dem die Fensterklasse in Windows registriert wird, zurück.

GetLine
```
BOOL GetLine(LPSTR ATextString, int StrSize,
             int LineNumber);
```

Holt eine Textzeile (deren Nummer angegeben ist) aus dem Editierfeld und liefert sie im nullterminierten String *ATextString* zurück. *StrSize* gibt an, wieviele Zeichen geholt werden sollen. *GetLine* liefert *FALSE* zurück, wenn kein Text gefunden werden kann oder der angegebene Puffer zu klein ist.

Querverweis: *TStatic::GetText, TEdit::GetNumLines, TEdit::GetLineLength*

GetLine-FromPos
```
int GetLineFromPos(int CharPos);
```

Liefert in einem mehrzeiligen Editierfeld die Nummer der Zeile, in der sich die in *CharPos* angegebene Zeichenposition befindet. Wenn *CharPos* größer als die Position des letzten Zeichens ist, wird die Nummer der letzten Zeile zurückgegeben. Wenn -1 angegeben ist, wird die Nummer der Zeile zurückgeliefert, die das erste ausgewählte Zeichen enthält.

GetLineIndex
```
int GetLineIndex(int LineNumber);
```

Liefert in einem mehrzeiligen Editierfeld die Anzahl der Zeichen, die vor der in *LineNumber* angegebenen Zeile stehen.

Wenn *LineNumber* -1 ist, wird die Nummer der Zeile zurückgeliefert, die das Caret enthält.

GetLineLength `int GetLineLength(int LineNumber);`

Liefert in einem mehrzeiligen Editierfeld die Anzahl der Zeichen in der Zeile, die in *LineNumber* angegeben ist. Wenn es sich dabei um -1 handelt und kein Text ausgefüllt ist, wird die Länge der Zeile zurückgeliefert, in der das Caret steht (die aktuelle Zeile). Ist Text in der Zeile ausgewählt, wird die Länge der Zeile abzüglich der ausgewählten Zeichen zurückgegeben. Wenn sich der ausgewählte Text über mehrere Zeilen erstreckt, wird die Länge der Zeilen minus der Anzahl der ausgewählten Zeichen zurückgeliefert.

GetNumLines `int GetNumLines();`

Liefert die Anzahl der Zeilen zurück, die in ein mehrzeiliges Editierfeld eingegeben wurden. Liefert Null zurück, wenn kein Text vorhanden oder ein Fehler aufgetreten ist.

GetSelection `void GetSelection(Rint StartPos, Rint EndPos);`

Holt die Anfangs- und Endposition des aktuell ausgewählten Textes und liefert sie in den Argumenten *StartPos* und *EndPos* zurück. Wenn Sie *GetSelection* zusammen mit *GetSubText* verwenden, können Sie den aktuell ausgewählten Text erhalten.

Querverweis: *TEdit::GetSubText*

GetSubText
```
void GetSubText(LPSTR ATextString, int StartPos,
                int EndPos);
```

Holt den Text in einem Editierfeld, der zwischen den Indizes *StartPos* und *EndPos* steht, und liefert ihn in *ATextString* zurück.

Querverweis: *TEdit::GetSelection*

Insert `void Insert(LPSTR ATextString);`

Fügt den in *ATextString* angegebenen Text in das Editierfeld an der aktuellen Caret-Position ein und ersetzt damit den aktuell ausgewählten Text. *Insert* ähnelt *Paste*, verwendet aber nicht die Zwischenablage.

Querverweis: *TEdit::Paste*

IsModified `BOOL IsModified();`

Liefert *TRUE* zurück, wenn der Anwender den Text im Editierfeld geändert hat.

Querverweis: *TEdit::ClearModify*

Paste `void Paste();`

Fügt Text aus der Zwischenablage an der aktuellen Position (Caret-Position) in das Editierfeld ein.

Querverweis: *TEdit::CMEditPaste*

Scroll
```
void Scroll(int HorizontalUnit, int VerticalUnit);
```
Führt in einem mehrzeiligen Editierfeld einen horizontalen oder vertikalen Bildlauf um die in *HorizontalUnit* und *VerticalUnit* angegebene Anzahl von Zeichen durch. Positive Werte bewirken einen Bildlauf nach rechts oder unten, negative Werte nach links oder oben.

Search
```
int Search(int StartPos, LPSTR AText,
           BOOL CaseSensitive);
```
Sucht in Abhängigkeit von *CaseSensitive* mit oder ohne Berücksichtigung der Groß-/Kleinschreibung nach dem angegebenen Text und wählt ihn aus, sobald er gefunden ist. Liefert die Position des Textes oder -1 zurück, falls kein Text gefunden wurde. Wenn in *StartPos* -1 angegeben ist, beginnt die Suche am Ende des aktuell ausgewählten Textes, oder an der Caret-Position, wenn kein Text ausgewählt ist.

SetSelection
```
BOOL SetSelection(int StartPos, int EndPos);
```
Bewirkt die Auswahl von Text zwischen den in *StartPos* und *EndPos* angegebenen Positionen, jedoch ohne das Zeichen an *EndPos* einzuschließen.

SetupWindow
```
virtual void SetupWindow();
```
protected

Wenn das Datenelement *TextLen* ungleich Null ist, wird die Anzahl der Zeichen, die in das Editierfeld eingegeben werden können, auf *TextLen* -1 beschränkt.

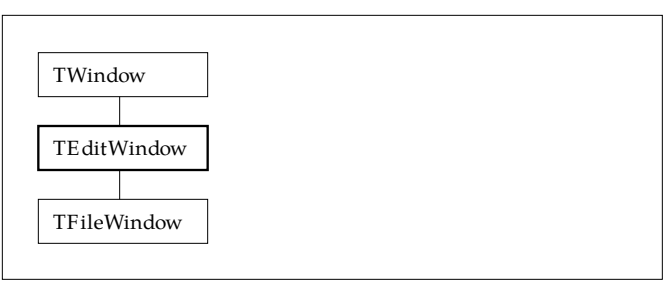
Querverweis: *TStatic::TextLen*

Undo
```
void Undo();
```
Macht den letzten Editiervorgang rückgängig.

Querverweis: *TEdit::CanUndo, TEdit::CMEditUndo*

TEditWindow

editwnd.h



TEditWindow definiert eine spezielle Fensterklasse, die das Eingeben und Editieren von Text ermöglicht. Sie enthält ein bildschirmgroßes Editierfenster, in dem die Textbearbeitung durchgeführt wird.

TFileWindow ist eine spezielle Ableitung von *TEditWindow*, die das Editieren von Dateien erlaubt.

Datenelemente

Editor `PEdit Editor;`

Editor zeigt auf ein bildschirmgroßes, mehrzeiliges Editierfeld, das dem Editierfenster die verfügbaren Textbearbeitungsmöglichkeiten zur Verfügung stellt.

IsReplaceOp `BOOL IsReplaceOp;`

Enthält *TRUE*, wenn mit dem nächsten Suchen ein Ersetzen verbunden ist.

SearchStruct `TSearchStruct SearchStruct;`

Ein Transferpuffer, der mit *TSearchDialog* verwendet wird.

Querverweis: *TSearchDialog*

Elementfunktionen

Konstruktor
```
TEditWindow(PTWindowsObject AParent, LPSTR ATitle,
            PTModule AModule = NULL);
```

Instantiiert das *Editor*-Editierfeld, füllt *SearchStruct* mit Null aus und setzt *IsReplaceOp* auf *NULL*. *ATitle* legt den Fenstertitel (in der Titelzeile) fest und wird an den *TWindow*-Konstruktor übergeben.

Konstruktor `TEditWindow(StreamableInit);` protected

Stream-Konstruktor von *TEditWindow*. Ruft den Stream-Konstruktor von *TWindow* auf. Wird dann aufgerufen, wenn ein *TEditWindow*-Objekt unter Verwendung von Daten aus einem Stream instantiiert wird.

build `static PTStreamable build();` protected

Ruft den Konstruktor *TEditWindow*(*StreamableInit*) auf und erstellt ein Objekt vom Typ *TEditWindow*, dessen Datenelemente anschließend aus einem Stream eingelesen werden.

CMEditFind
```
virtual void CMEditFind(RTMessage Msg) =
[CM_FIRST + CM_EDITFIND];
```
protected

Antwortet auf die Auswahl eines *Suchen*-Menüeintrags. Führt einen *Suchen*-Dialog aus, um den vom Anwender eingegebenen Text und die Optionen zu ermitteln (die in *SearchStruct*

übertragen werden). Setzt *IsReplaceOp* auf *FALSE* und ruft dann *DoSearch* auf.

Querverweis: *TEditWindow::DoSearch*, *TSearchStruct*

CMEditFind-Next

```
virtual void CMEditFindNext(RTMessage Msg) =
[CM_FIRST + CM_EDITFINDNEXT];
```
protected

Antwortet durch den Aufruf von *DoSearch* auf die Auswahl eines *Weitersuchen*-Menüeintrags. Dabei wird davon ausgegangen, daß *SearchStruct* bereits Text und *Suchen-/Ersetzen*-Optionen enthält.

Querverweis: *TEditWindow::DoSearch*, *TSearchStruct*

CMEditReplace

```
virtual void CMEditReplace(RTMessage Msg) =
[CM_FIRST + CM_EDITREPLACE];
```
protected

Antwortet durch den Aufruf von *DoSearch* auf die Auswahl eines *Ersetzen*-Menüeintrags. Führt einen *Suchen*-Dialog aus, um den vom Anwender eingebenenen Text und die Optionen zu ermitteln (die in *SearchStruct* übertragen werden). Informationen dazu finden Sie unter *TSearchStruct* in Kapitel 24 »Weitere Komponenten von ObjectWindows«. Setzt *IsReplaceOp* auf *TRUE* und ruft dann *DoSearch* auf.

Querverweis: *TEditWindow::DoSearch*

DoSearch

```
void DoSearch();
```

Sucht in *Editor* nach dem Text und mit den Optionen, die in *SearchStruct* angegeben sind. Dabei werden Meldungsfenster eingeblendet, die entweder Fehler anzeigen oder Bestätigungen für Ersetzungen anfordern (falls gewünscht). Informationen dazu finden Sie unter *TSearchStruct* in Kapitel 24 »Weitere Komponenten von ObjectWindows«.

Querverweis: *TEdit::Search*

read

```
virtual Pvoid read(Ripstream is);
```
protected

Ruft *TWindow::read* auf, um das Basisobjekt *TWindow* einzulesen. Danach erfolgt ein Aufruf von *GetChildPtr*, um das Editierfeld einzulesen, das ein untergeordnetes Fenster ist.

WMSetFocus

```
virtual void WMSetFocus(RTMessage Msg) =
[WM_FIRST + WM_SETFOCUS];
```
protected

Antwortet auf eine eingehende *WM_SETFOCUS*-Botschaft und übergibt dabei den Fokus an das *Editor*-Editierfeld.

WMSize

```
virtual void WMSize(RTMessage Msg) =
[WM_FIRST + WM_SIZE];
```
protected

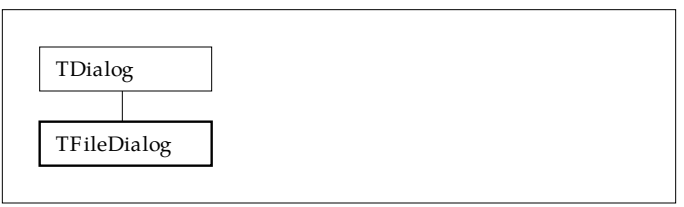
Antwortet auf eine eingehende *WM_SIZE*-Botschaft durch Anpassung der Größe des *Editor*-Editierfelds an die Größe des *TEditWindow*-Client-Bereichs.

write `virtual void write(Ropstream os);` protected

Ruft *TWindow::write* auf, um das Basisobjekt *TWindow* in den Stream zu schreiben. Danach erfolgt ein Aufruf von *PutChildPtr*, um das Editierfeld, das ein untergeordnetes Fenster ist, an den Stream zu übergeben.

TFileDialog

filedial.h



TFileDialog erstellt Dialoge, die den Anwender zur Auswahl eines Dateinamens für das Öffnen oder Sichern einer Datei auffordern. In der Dialogressource FILEDIAL.DLG stehen zwei Dialogressourcen-Definitionen – für einen Dialog zum Öffnen und einen Dialog zum Sichern von Dateien – zur Verfügung.

Datenelemente

Extension `char Extension[MAXEXT];`

Enthält die Dateinamenserweiterung.

FilePath `LPSTR FilePath;`

Zeigt auf den Puffer, der den vom Anwender angegebenen Dateinamen zurückliefert. Wenn dieses Element an den Konstruktor des Dialogs übergeben wird, enthält es die Dateimaske, die zu Beginn angezeigt wird.

FileSpec `char FileSpec[FILESPEC];`

Enthält den aktuellen Dateinamen.

PathName `char PathName[MAXPATH];`

Enthält den aktuellen Pfadnamen.

Elementfunktionen

Konstruktor
```
TFileDialog(PTWindowsObject AParent, int ResourceId,
            LPSTR AFilePath, PTModule AModule = NULL);
```

Ruft den Konstruktor von *TDialog* auf und übergibt *AParent*, *ResourceId* und *AModule*. Setzt *FilePath* auf *AFilePath*. Verwendet den Inhalt des Puffers, auf den *AFilePath* zeigt, als Vorlage für das Ausfüllen der Datei-Liste.

Geben Sie entweder *SD_FILEOPEN* oder *SD_FILESAVE* als *ResourceId* (Ressourcen-Bezeichner vom Typ *Integer*) für den Dialog an.

Querverweis: *TDialog::TDialog*

Konstruktor `TFileDialog(StreamableInit);` protected

Stream-Konstruktor von *TFileDialog*. Wird aufgerufen, wenn ein *TFileDialog*-Objekt unter Verwendung von Daten aus einem Stream instantiiert wird.

build `static PTStreamable build();`

Ruft den Konstruktor *TFileDialog*(*StreamableInit*) auf und erstellt ein Objekt vom Typ *TFileDialog*, dessen Datenelemente anschließend aus einem Stream eingelesen werden.

CanClose `virtual BOOL CanClose();`

Liefert *TRUE* zurück, wenn der Anwender einen gültigen Dateinamen eingegeben hat, holt den Text des Editierfeldes und aktualisiert *PathName*.

Die Funktion ruft *UpdateListBoxes* auf; sie liefert *FALSE*, wenn *UdateListBoxes TRUE* zurückgibt, oder wenn das Editierfeld einen ungültigen Dateinamen enthält.

Querverweis: *TFileDialog::UpdateListBoxes*

HandleDList `virtual void HandleDList(RTMessage Msg) = [ID_FIRST + ID_DLIST];` protected

Antwortet auf Botschaften der Verzeichnis-Liste und aktualisiert *PathName*, wenn sich die Auswahl in der Liste ändert. Die Funktion ruft *UpdateListBoxes* auf, wenn der ausgewählte Eintrag doppelt angeklickt wurde; andernfalls wird *UpdateFileName* aufgerufen.

Sie löscht die Auswahl in der Liste, wenn der Fokus weitergegeben wird.

Querverweis: *TFileDialog::UpdateListBoxes*

HandleFList `virtual void HandleFList(RTMessage Msg) = [ID_FIRST + ID_FLIST];` protected

Antwortet auf Botschaften der Datei-Liste und aktualisiert *PathName* mit dem Namen der ausgewählten Datei. Die Funktion ruft *UpdateFileName* auf, wenn sich die Auswahl in der Liste ändert.

Sie versucht den Dialog zu schließen, wenn der ausgewählte Eintrag doppelt angeklickt wurde. Sie löscht die Auswahl in der Liste, wenn der Fokus weitergegeben wird.

Querverweis: *TFileDialog::UpdateFileName*

HandleFName
```
virtual void HandleFName(RTMessage Msg) =
[ID_FIRST + ID_FNAME];
```
protected

Antwortet auf Botschaften des Editierfeldes und aktiviert den *OK*-Schalter, wenn das Editierfeld Text enthält.

SelectFileName
```
void SelectFileName();
```

Wählt den Text im Editierfeld aus und setzt den Fokus auf das Editierfeld.

SetupWindow
```
virtual void SetupWindow();
```
protected

Erstellt das Datei-Dialogfenster durch einen Aufruf von *TDialog::SetupWindow*. Die Funktion begrenzt die Anzahl der Zeichen, die in das Editierfeld eingegeben werden können, auf *MAXPATH*. Danach werden die Funktionen *UpdateListBoxes* und *SelectFileName* aufgerufen.

Querverweis: *TDialog::SetupWindow*, *TDialog::UpdateListBoxes*, *TFileDialog::SelectFileName*

UpdateFileName
```
void UpdateFileName();
```

Setzt den Text des Editierfeldes auf *PathName* und wählt den Text aus.

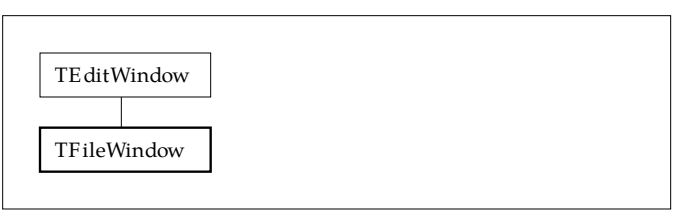
UpdateListBoxes
```
BOOL UpdateListBoxes();
```

Versucht beim Öffnen einer Datei die Datei- und die Verzeichnis-Liste, beim Sichern einer Datei nur die Verzeichnis-Liste zu aktualisieren. Bei erfolgreicher Aktualisierung wird *UpdateFileName* aufgerufen und *TRUE* zurückgeliefert, andernfalls *FALSE*.

Querverweis: *TFileDialog::UpdateFileName*

TFileWindow

filewnd.h



TFileWindow ist ein Datei-Editierfenster, das auf *TEditWindow* basiert. Dateifenster verwenden Dialoge zum Öffnen und Sichern von Dateien. Die Datenelemente und Elementfunktionen von *TFileWindow* implementieren Datei-Dialoge und Botschaftsantwortfunktionen für die gebräuchlichen Menübefehle *Open* (*Öffnen*), *Read* (*Lesen*), *Write* (*Schreiben*), *Save* (*Speichern*) und *SaveAs* (*Speichern unter*).

Datenelemente

FileName
```
LPSTR FileName;
```
Enthält den Namen der Datei, die editiert werden soll. Dieses Datenelement wird vom Dialog und den Öffnen- und Speichern-Funktionen verwendet.

IsNewFile
```
BOOL IsNewFile;
```
IsNewFile ist *TRUE*, wenn der Anwender eine neue Datei editiert, und *FALSE*, wenn der Anwender eine vorher geöffnete Datei editiert.

Elementfunktionen

Konstruktor
```
TFileWindow(PTWindowsObject AParent, LPSTR ATitle,
            LPSTR AFileName, PTModule AModule = NULL);
```
Ruft einen *TEditWindow*-Konstruktor auf und übergibt *AParent* und *ATitle*. Setzt *FileName* auf eine Kopie von *AFileName* und *IsNewFile* auf *TRUE*.

Konstruktor
```
TFileWindow(StreamableInit);
```
protected

Stream-Konstruktor von *TFileWindow*. Ruft den Stream-Konstruktor von *TEditWindow* auf. Wird aufgerufen, wenn ein *TFileWindow*-Objekt unter Verwendung von Daten aus einem Stream instantiiert wird.

Destruktor
```
virtual ~TFileWindow();
```
Gibt den für die Speicherung des *TFileWindow*-Namens reservierten Platz frei.

build
```
static PTStreamable build();
```
Ruft den Konstruktor *TFileWindow*(*StreamableInit*) auf und erstellt ein Objekt vom Typ *TFileWindow*, dessen Datenelemente anschließend aus einem Stream eingelesen werden.

CanClear
```
virtual BOOL CanClear();
```
Liefert einen *BOOL*-Wert zurück, der angibt, ob der Text des *Editor*-Editierfeldes gelöscht werden darf oder nicht. Die Funktion liefert *TRUE* zurück, wenn sich der Text nicht geändert hat.

Wurde der Text geändert, wird der Anwender aufgefordert, die Datei zu sichern oder das Löschen des Textes zu bestätigen.

CanClose
```
virtual BOOL CanClose();
```
Liefert *TRUE*, wenn *TFileWindow* geschlossen werden kann. Dabei wird das Ergebnis eines *CanClear*-Aufrufs zurückgegeben.

CMFileNew
```
virtual void CMFileNew(RTMessage Msg) =
[CM_FIRST + CM_FILENEW];                    protected
```

Antwortet mit einem Aufruf von *NewFile* auf einen eingehenden Befehl *Datei Neu* (eine Botschaft mit einem *CM_FILENEW*-Befehlsbezeichner).

Querverweis: *TFileWindow::NewFile*

CMFileOpen
```
virtual void CMFileOpen(RTMessage Msg) =
[CM_FIRST + CM_FILEOPEN];                   protected
```

Antwortet mit einem Aufruf von *Open* auf einen eingehenden Befehl *Datei Öffnen* (eine Botschaft mit einem *CM_FILEOPEN*-Befehlsbezeichner).

Querverweis: *TFileWindow::Open*

CMFileSave
```
virtual void CMFileSave(RTMessage Msg) =
[CM_FIRST + CM_FILESAVE];                   protected
```

Antwortet mit einem Aufruf von *Save* auf einen eingehenden Befehl *Datei Speichern* (eine Botschaft mit einem *CM_FILESAVE*-Befehlsbezeichner).

Querverweis: *TFileWindow::Save*

CMFileSaveAs
```
virtual void CMFileSaveAs(RTMessage Msg) =
[CM_FIRST + CM_FILESAVEAS];                 protected
```

Antwortet mit einem Aufruf von *SaveAs* auf einen eingehenden Befehl *Datei Speichern unter* (eine Botschaft mit einem *CM_FILESAVEAS*-Befehlsbezeichner).

Querverweis: *TFileWindow::SaveAs*

NewFile
```
void NewFile();
```

Beginnt mit dem Editieren einer neuen Datei, nachdem *CanClear* aufgerufen wurde, um festzustellen, ob der Text des *Editor*-Editierfeldes gelöscht werden kann.

Querverweis: *TFileWindow::CanClear*

Open
```
void Open();
```

Öffnet eine neue Datei, nachdem der Anwender bestätigt hat, daß der Text des *Editor*-Editierfeldes gelöscht werden darf. Die Funktion ruft *CanClear* auf, und zeigt – wenn *TRUE* zurückgeliefert wird – ein Dialogfenster an, das den Anwender zur Eingabe eines Namens für die neue Datei auffordert. Der neue Dateiname wird dann beim Aufruf von *ReplaceWith* an diese Funktion übergeben.

Querverweis: *TFileWindow::CanClear, TFileWindow::ReplaceWith*

Read `BOOL Read();`

Liest den Inhalt einer zuvor angegeben Datei in das *Editor*-Objekt. Die Funktion setzt *IsNewFile* auf *FALSE* und liefert *TRUE* zurück, wenn die Operation erfolgreich war.

read `virtual Pvoid read(Ripstream is);`

Ruft *TEditWindow::read* auf, um das Basisobjekt *TEditWindow* einzulesen. Danach wird *FileName* gelesen und *IsNewFile* auf *TRUE* gesetzt, wenn *FileName* gleich *NULL* ist. Andernfalls wird *IsNewFile* auf *FALSE* gesetzt.

ReplaceWith `void ReplaceWith(LPSTR AFileName);`

Ersetzt die aktuell editierte Datei durch eine Datei mit angegebenen Namen. Dazu werden *SetFileName* und *Read* aufgerufen.

Querverweis: *TFileWindow::SetFileName, TFileWindow::Read*

Save `BOOL Save();`

Speichert den Inhalt des *Editor*-Objekts in eine Datei. Wenn *Editor->IsModified FALSE* zurückliefert, gibt *Save TRUE* zurück. Damit wird angezeigt, daß keine Änderungen durchgeführt wurden oder der geänderte Inhalt bereits abgespeichert wurde. Wenn *IsNewFile TRUE* ist, wird *SaveAs* aufgerufen, andernfalls erfolgt ein Aufruf von *Write*. Die Funktion liefert das Ergebnis des jeweiligen Aufrufs zurück.

Querverweis: *TFileWindow::SaveAs, TFileWindow::Write*

SaveAs `BOOL SaveAs();`

Speichert den Inhalt des Datenelements *Editor* in die Datei, deren Name über die Ausführung eines »Datei Speichern«-Dialogfensters vom Anwender angefordert wird. Wenn der Anwender den Dialog mit *OK* beendet, ruft die Funktion *SetFileName* und *Write* auf. Schließlich wird *TRUE* zurückgeliefert, wenn die Datei gespeichert ist.

Querverweis: *TFileWindow::SetFileName, TFileWindow::Write*

SetFileName `void SetFileName(LPSTR AFileName);`

Setzt *FileName* und aktualisiert den Titel des Fensters.

SetupWindow `virtual void SetupWindow();` protected

Erstellt durch den Aufruf von *TEditWindow::SetupWindow* das Editierfeld *Editor*. Setzt den Fenstertitel auf *FileName* (wenn vorhanden), andernfalls auf "Untitled".

Querverweis: *TFileWindow::SetFileName, TFileWindow::Read*

Write `BOOL Write();` protected

Speichert den Inhalt des Datenelements *Editor* in die Datei, deren Name in *FileName* angegeben ist.

Die Funktion liefert *TRUE* zurück, wenn die Operation erfolgreich war.

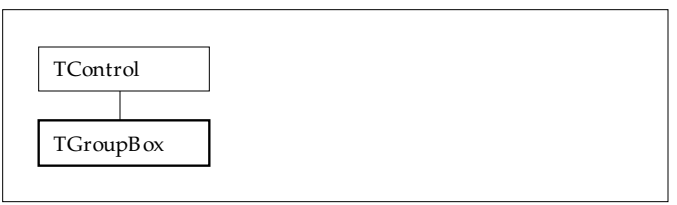
write `virtual void write(Ropstream os);` protected

Aktiviert *TEditWindow::write*, um das Basisobjekt *TEditWindow* in den Stream zu schreiben. Danach wird *FileName* an den Stream übergeben.

Die Funktion liefert *TRUE*, wenn die Operation erfolgreich war.

TGroupBox

groupbox.h



Eine Instanz von *TGroupBox* ist ein Schnittstellenobjekt, das das entsprechende Gruppen-Element in Windows repräsentiert. *TGroupBox*-Objekte werden normalerweise nicht in Dialogen oder Dialogfenstern (*TDialog*) verwendet, sondern dann eingesetzt, wenn Sie eine Gruppe in einem Fenster erstellen wollen.

Gruppen sollen primär nicht irgendwelche Aktionen auslösen; ihr Zweck besteht vielmehr darin, eine Gruppe von Wahlfeldern (Auswahl- und Schaltfeldern) oder anderen Steuerelementen optisch zusammenzufassen.

Im Hintergrund können sie jedoch eine wichtige Rolle bei der Behandlung von Status-Änderungen ihrer Steuerelemente (normalerweise Auswahl- und Schaltfelder) spielen.

Angenommen, Ihr Programm soll auf Auswahländerungen in einer der Schalter-Gruppen immer auf ähnliche Weise antworten. Leiten Sie dazu eine Klasse von *TGroupBox* ab, die die Elementfunktion *SelectionChanged* überschreibt.

Alternativ könnten Sie auf Auswahländerungen in der Gruppe der Schaltfelder auch mit der Definition einer Antwort für das übergeordnete Fenster der Gruppe antworten.

Dazu definieren Sie eine auf der ID des untergeordneten Fensters basierende Botschaftsantwortfunktion, die die ID der Gruppe verwendet. Die Gruppe sendet automatisch eine auf der ID des untergeordneten Fensters basierende Botschaft an ihr übergeordnetes Fenster, sobald ein anderes Schaltfeld gewählt wird.

Datenelement

NotifyParent
```
BOOL NotifyParent;
```
Ein Flag, das anzeigt, ob das übergeordnete Fenster benachrichtigt werden soll, wenn sich der Status der Wahlfelder in der Gruppe geändert hat.

Voreingestellt ist *TRUE*.

Element-funktionen

Konstruktor
```
TGroupBox(PTWindowsObject AParent, int AnId,
          LPSTR AText, int X, int Y, int W, int H,
          PTModule AModule = NULL);
```
Erstellt ein *TGroupBox*-Objekt mit dem übergeordneten Fenster (*AParent*), der Steuerelement-ID (*AnId*), dem zugehörigen Text (*AText*), mit der Position (*X*, *Y*) relativ zum Ursprung des Client-Bereichs des übergeordneten Fensters, mit der Breite (*W*) und der Höhe (*H*).

Ruft den *TControl*-Konstruktor mit entsprechenden Parametern auf und ändert dann *Attr.Style* durch Hinzufügen von *BS_GROUPBOX* und Entfernen von *WS_TABSTOP*. *NotifyParent* ist auf *TRUE* gesetzt; per Voreinstellung wird das übergeordnete Fenster der Gruppe benachrichtigt, wenn sich in einem der Steuerelemente der Gruppe die Auswahl ändert.

Querverweis: *TControl::TControl*

Konstruktor
```
TGroupBox(PTWindowsObject AParent, int ResourceId,
          PTModule AModule = NULL);
```
Erstellt ein *TGroupBox*-Objekt, das einem Gruppen-Steuerelement eines *TDialog*-Objekts zugeordnet werden soll, und ruft den *TControl*-Konstruktor mit identischen Parametern auf. Danach wird *DisableTransfer* aufgerufen, um die Gruppe vom Trasfermechanismus auszuschließen, da keine zu übertragenden Daten vorhanden sind.

Der Parameter *ResourceId* muß einem Bezeichner für eine Gruppenressource entsprechen, die Sie definieren.

Querverweis: *TControl::TControl, TWindowsObject::DisableTransfer*

Konstruktor
```
TGroupBox(StreamableInit);
```
protected

Stream-Konstruktor von *TGroupBox*.

Ruft den Stream-Konstruktor von *TControl* auf. Er wird aufgerufen, wenn ein *TGroupBox*-Objekt unter Verwendung von Daten aus einem Stream instantiiert wird.

Querverweis: *TControl::TControl*

build `static PTStreamable build();`

Ruft den Konstruktor *TGroupBox*(*StreamableInit*) auf und erstellt ein Objekt vom Typ *TGroupBox*, dessen Datenelemente anschließend aus einem Stream eingelesen werden.

GetClassName `virtual LPSTR GetClassName();`

Liefert "BUTTON", den Bezeichner, unter dem die Fensterklasse in Windows registriert wird, zurück.

read `virtual Pvoid read(Ripstream is);` protected

Ruft *TWindow::read* auf, um das Basisobjekt *TGroupBox* einzulesen. Danach wird *NotifyParent* eingelesen.

SelectionChanged `virtual void SelectionChanged(int ControlId);`

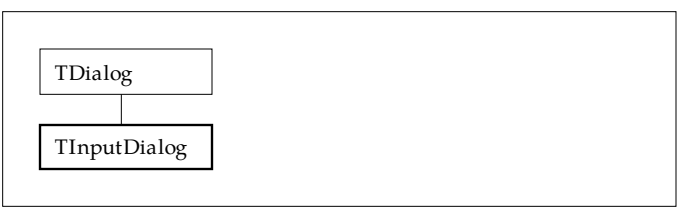
Wenn *NotifyParent TRUE* ist, benachrichtigt diese Funktion das übergeordnete Fenster davon, daß sich eine Auswahl in der Gruppe geändert hat; zu diesem Zweck wird eine auf der ID des untergeordneten Fensters basierende Botschaft gesendet. Diese Elementfunktion könnte überschrieben werden, so daß die Gruppe Auswahländerungen in ihren Steuerelement-Gruppen bearbeitet.

write `virtual void write(Ropstream os);` protected

Ruft *TWindow::write* auf, um das Basisobjekt *TGroupBox* in den Stream zu schreiben. Danach wird *NotifyParent* an den Stream übergeben.

TInputDialog

inputdia.h



TInputDialog stellt einen generischen Dialog zur Verfügung, der vom Anwender eine Texteingabe verlangt. Bei der Einstellung werden Titel, Eingabeaufforderung und voreingestellter Eingabetext festgelegt.

Datenelemente

Buffer `LPSTR Buffer;`

Ein Zeiger auf den Puffer, der den vom Anwender angeforderten Text zurückliefert. Wenn *Buffer* an den Konstruktor des

Eingabe-Dialogs übergeben wird, enthält es den Text, der zu Beginn im Editierfeld angezeigt wird.

BufferSize

```
WORD BufferSize;
```

Enthält die Größe des Puffers, der die Anwendereingabe aufnimmt.

Prompt

```
LPSTR Prompt;
```

Zeigt auf die Eingabeaufforderung des Eingabe-Dialogs.

Element-funktionen

Konstruktor

```
TInputDialog(PTWindowsObject AParent, LPSTR ATitle,
             LPSTR APrompt, LPSTR ABuffer, WORD
             ABufferSize, PTModule AModule = NULL);
```

Ruft den Konstruktor von *TDialog* auf und übergibt *AParent*, den Ressourcenbezeichner, *SD_INPUTDIALOG* und *AModule*. Setzt den Titel des Dialogs auf *ATitle* und das statische Eingabeaufforderungs-Steuerelement auf *APrompt*.

Danach werden die Datenelemente *Buffer* und *BufferSize* auf *ABuffer* und *ABufferSize* gesetzt.

Querverweis: *TDialog::TDialog*

Konstruktor

```
TInputDialog(StreamableInit);
```

protected

Stream-Konstruktor von *TInputDialog*. Ruft den Stream-Konstruktor von *TDialog* auf. Er wird aufgerufen, wenn ein *TInputDialog*-Objekt unter Verwendung von Daten aus einem Stream instantiiert wird.

Querverweis: *TDialog::TDialog*

build

```
static PTStreamable build();
```

Ruft den Konstruktor *TInputDialog*(*StreamableInit*) auf und erstellt ein Objekt vom Typ *TInputDialog*, dessen Datenelemente anschließend aus einem Stream eingelesen werden.

read

```
virtual Pvoid read(Ripstream is);
```

protected

Ruft *TDialog::read* auf, um das Basisobjekt *TDialog* einzulesen. Danach wird *Prompt* eingelesen.

SetupWindow

```
virtual void SetupWindow();
```

protected

Baut durch einen Aufruf von *TDialog::SetupWindow* das Fenster auf und begrenzt dann die Anzahl der Zeichen, die der Anwender eingeben kann, auf *BufferSize* -1.

TransferData

```
void TransferData(WORD Direction);
```

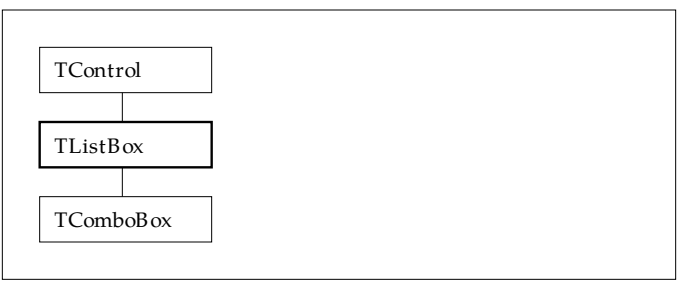
Überträgt die Daten des Eingabe-Dialogs. Wenn *Direction* auf *TF_SETDATA* gesetzt ist, wird der Text des statischen Steuerelements und des Editierfeldes im Dialog auf den Text in *Prompt*

und *Buffer* gesetzt. Ist *Direction* auf *TF_GETDATA* gesetzt, wird *Buffer* mit dem aktuellen Text in *Editor* ausgefüllt.

write `virtual void write(Ropstream os);` protected

Ruft *TDialog::write* auf, um das Basisobjekt *TDialog* in den Stream zu schreiben. Danach wird *Prompt* an den Stream übergeben.

TListBox listbox.h



TListBox ist ein Schnittstellenobjekt, das ein entsprechendes Listenelement in Windows repräsentiert. Für die Erstellung einer Liste in einem übergeordneten *TWindow*-Fenster muß *TListBox* verwendet werden. *TListBox* vereinfacht auch die Kommunikation zwischen Ihrer Anwendung und den Listen eines *TDialog*-Objekts. Die Elementfunktionen von *TListBox* dienen auch den Instanzen der davon abgeleiteten Klasse *TComboBox*.

Element-funktionen

Konstruktor
```
TListBox(PTWindowsObject AParent, int AnId, int X,
         int Y, int W, int H, PTModule AModule = NULL);
```

Erstellt ein Listenobjekt mit dem angegebenen übergeordneten Fenster (*AParent*), der Steuerelement-ID (*AnId*), mit der Position (*X*, *Y*), die relativ zum Ursprungspunkt des Client-Bereichs des übergeordneten Fensters ist, mit der Breite (*W*) und der Höhe (*H*). Die Funktion ruft einen *TControl*-Konstruktor auf, fügt *LBS_STANDARD* zu den voreingestellten Stilen für die Liste hinzu und stellt damit folgendes zur Verfügung:

- einen Rahmen (*WS_BORDER*),
- eine vertikale Bildlaufleiste (*WS_VSCROLL*),
- alphabetische Sortierung der Listeneinträge (*LBS_SORT*),
- eine Benachrichtigung des übergeordneten Fensters über Auswahlen (*LBS_NOTIFY*).

Konstruktor
```
TListBox(PTWindowsObject AParent, int ResourceId,
         PTModule AModule = NULL);
```

Erstellt ein *TListBox*-Objekt, das mit einem Listen-Steuerelement eines *TDialog*-Objekts verknüpft werden soll und ruft den *TControl*-Konstruktor mit entsprechenden Parametern auf.

Der Parameter *ResourceId* muß einer Listenressource entsprechen, die Sie definieren.

Querverweis: *TControl::TControl*

Konstruktor
```
TListBox(StreamableInit);
```
protected

Stream-Konstruktor von *TListBox*. Ruft den Stream-Konstruktor von *TControl* auf. Er wird aufgerufen, wenn ein *TListBox*-Objekt unter Verwendung von Daten aus einem Stream instantiiert wird.

Querverweis: *TControl::TControl*

AddString
```
int AddString(LPSTR AString);
```

Fügt *AString* in die Liste ein und liefert dabei dessen Position in der Liste zurück (Null ist die erste Position). Die Funktion liefert einen negativen Wert, wenn ein Fehler auftritt. Die Listeneinträge werden automatisch sortiert. Eine Sortierung unterbleibt, wenn der Stil *LBS_SORT* vor der Erstellung aus dem Datenelement *Attr.Style* des Listenobjekts entfernt wurde.

build
```
static PTStreamable build();
```

Ruft den Konstruktor *TListBox*(*StreamableInit*) auf und erstellt ein Objekt vom Typ *TListBox*, dessen Datenelemente anschliessend aus einem Stream eingelesen werden.

ClearList
```
void ClearList();
```

Löscht alle Einträge in der Liste.

DeleteString
```
int DeleteString(int Index);
```

Löscht den Eintrag in der Liste an der Position, die in *Index* angegeben ist (die Positionszählung beginnt bei 0). *DeleteString* liefert die Anzahl der verbleibenden Listeneinträge oder einen negativen Wert zurück, wenn ein Fehler auftritt.

FindExactString
```
int FindExactString(LPSTR AString, int SearchIndex);
```

Durchsucht die Liste – beginnend in der Zeile mit der in *SearchIndex* übergebenen Zeilenummer – nach einem String, der genau dem String *AString* entspricht. Konnte nach dem letzten Stringvergleich keine Entsprechung festgestellt werden, wird die Suche vom Anfang der Liste an durchgeführt, bis eine Entsprechung gefunden wird oder die Liste vollständig durchsucht ist. Die Funktion startet die Suche am Beginn der

Liste, wenn in *SearchIndex* -1 angegeben ist. Sie liefert bei erfolgreicher Suche den Index des ersten gefundenen Strings zurück bzw. einen negativen Wert, wenn ein Fehler aufgetreten ist.

FindString `int FindString(LPSTR AString, int SearchIndex);`

Durchsucht die Liste wie bei *FindExactString* beschrieben, sucht dabei aber nach dem ersten Eintrag, der mit *AString* beginnt.

GetClassName `virtual LPSTR GetClassName();` protected

Liefert "LISTBOX", den Bezeichner, unter dem die Fensterklasse in Windows registriert wird, zurück.

GetCount `int GetCount();`

Liefert die Anzahl der Einträge in der Liste zurück, oder einen negativen Wert, wenn ein Fehler auftritt.

GetMsgID `virtual WORD GetMsgID(WORD AMsg);` protected

Liefert den Bezeichner der Windows-Listenbotschaft zurück, der mit dem angegebenen ObjectWindows-Botschaftsbezeichner verknüpft ist. Die Funktion ermöglicht, daß *TComboBox* (und davon abgeleitete Klassen) viele der *TListBox*-Elementfunktionen verwenden können.

GetSelCount `int GetSelCount();`

Liefert die Anzahl der ausgewählten Einträge in der Liste zurück (für Listen mit einer oder mehreren Auswahlmöglichkeiten und für Kombinationsfenster).

GetSelIndex `int GetSelIndex();`

Liefert den positiven Index (beginnend bei Null) des aktuell ausgewählten Eintrags zurück bzw. einen negativen Wert, wenn kein Eintrag ausgewählt ist (für Listen mit einer Auswahlmöglichkeit und Kombinationsfenster).

GetSelIndexes `int GetSelIndexes(Pint Indexes, int MaxCount);`

Für Listen mit mehreren Auswahlmöglichkeiten. Füllt *Indexes* mit den Indizes der ausgewählten Einträge (maximal *MaxCount*). Liefert die Anzahl der Einträge zurück, die in *Indexes* eingebracht wurden (-1 für Listen mit einer Auswahlmöglichkeit und für Kombinationsfenster).

GetSelString `int GetSelString(LPSTR AString, int MaxChars);`

Holt den aktuell ausgewählten Eintrag bis zu einer maximalen Länge von *MaxChars* und liefert ihn in *AString* zurück (einschließlich einer abschließenden Null). *GetSelString* liefert die Länge des Strings bzw. einen negativen Wert zurück, wenn ein Fehler aufgetreten ist. Wenn kein String ausgewählt ist, wird

-1 zurückgegeben (für Listen mit einer Auswahlmöglichkeit und Kombinationsfenster).

GetSelStrings
```
int GetSelStrings(LPSTR *Strings, int MaxCount,
                  int MaxChars);
```

Für Listen mit mehreren Auswahlmöglichkeiten. Holt die aktuell ausgewählten Einträge und bringt *MaxCount* von ihnen in *Strings* ein. Jede Komponente des Arrays *Strings* sollte *MaxChars* Zeichen plus ein abschließendes Nullzeichen aufnehmen können. Diese Funktion liefert die Anzahl der Einträge zurück, die in *Strings* eingebracht wurden (-1 für Listen mit einer Auswahlmöglichkeit und für Kombinationsfenster).

GetString
```
int GetString(LPSTR AString, int Index);
```

Holt den Eintrag an der in *Index* angegebenen Position (die Positionszählung beginnt bei Null) und liefert ihn in *AString* zurück. *GetString* liefert die Länge des Strings bzw. einen negativen Wert zurück, wenn ein Fehler auftritt.

GetStringLen
```
int GetStringLen(int Index);
```

Liefert die Stringlänge des Eintrags (ohne die abschließende Null) an der in *Index* angegebenen Indexposition zurück, bzw. einen negativen Wert, wenn ein Fehler auftritt.

InsertString
```
int InsertString(LPSTR AString, int Index);
```

Fügt *AString* an der in *Index* angegebenen Position in der Liste ein und liefert die aktuelle Position des Eintrags in der Liste zurück. Die Positionszählung beginnt bei Null. Tritt ein Fehler auf, wird ein negativer Wert zurückgeliefert. Die Liste wird nicht erneut sortiert. Wenn *Index* -1 ist, wird der String an das Ende der Liste angefügt.

SetSelIndex
```
int SetSelIndex(int Index);
```

Bewirkt die Auswahl des Eintrags an der in *Index* angegebenen Position. Die Positionszählung beginnt bei Null. Wenn *Index* -1 enthält, wird jede Auswahl in der Liste aufgehoben.

SetSelIndex liefert eine negative Zahl, wenn ein Fehler auftritt (für Listen mit einer Auswahlmöglichkeit und für Kombinationsfenster).

SetSelIndexes
```
int SetSelIndexes(Pint Indexes, int NumSelections,
                  BOOL ShouldSet);
```

Für Listen mit mehreren Auswahlmöglichkeiten. Wählt die Strings an den im Array *Indexes* angegeben Indizes aus oder macht die Auswahl rückgängig.

Wenn *ShouldSet TRUE* ist, werden die indizierten Strings ausgewählt und markiert. Wenn *ShouldSet FALSE* ist, wird die Hervorhebung deaktiviert und die Strings sind nicht mehr

ausgewählt. Diese Funktion liefert die Anzahl der Strings zurück, deren Auswahl erfolgreich durchgeführt oder rückgängig gemacht wurde (-1 für Listen mit einer Auswahlmöglichkeit und für Kombinationsfenster).

Wenn *NumSelections* kleiner als Null ist, wird für alle Strings die Auswahl aktiviert oder deaktiviert. Bei Auftreten eines Fehlers wird ein negativer Wert zurückgeliefert.

SetSelString

```
int SetSelString(LPSTR AString, int AIndex);
```

Bewirkt die Auswahl des ersten Eintrags, der mit dem Text in *AString* beginnt und der hinter der in *AIndex* angegebenen Position liegt. Die Positionszählung beginnt bei Null.

Wenn *AIndex* -1 enthält, wird die gesamte Liste – ab dem ersten Eintrag – durchsucht. *SetSelString* liefert die Position des neu ausgewählten Eintrags zurück, bzw. einen negativen Wert, wenn ein Fehler auftritt (für Listen mit einer Auswahlmöglichkeit und für Kombinationsfenster).

SetSelStrings

```
int SetSelStrings(LPSTR *Prefixes, int NumSelections,
                  BOOL ShouldSet);
```

Für Listen mit mehreren Auswahlmöglichkeiten. Wählt die Strings in der zugehörigen Liste aus, die mit den angegebenen Präfixen beginnen. Die Suche nach jedem String beginnt am Anfang der Liste und wird so lange weitergeführt, bis eine Entsprechung gefunden wird oder die Liste vollständig abgearbeitet ist.

Wenn *ShouldSet TRUE* ist, werden die gefundenen übereinstimmenden Strings ausgewählt und markiert. Ist *ShouldSet FALSE*, wird die Hervorhebung deaktiviert und die Strings sind nicht länger ausgewählt. Diese Funktion liefert die Anzahl der Strings zurück, deren Auswahl erfolgreich aktiviert oder deaktiviert wurde (-1 für Listen mit einer Auswahlmöglichkeit und für Kombinationsfenster).

Wenn *NumSelections* kleiner als Null ist, wird für alle Strings die Auswahl aktiviert oder deaktiviert. Bei Auftreten eines Fehlers wird ein negativer Wert zurückgeliefert.

Transfer

```
virtual WORD Transfer(Pvoid DataPtr, WORD TransferFlag);
```

Überträgt die Einträge und Auswahlen der Liste in einen oder aus einem Transferpuffer, wenn *TransferFlag* entweder auf *TF_SETDATA* oder *TF_GETDATA* gesetzt ist. *TransferBuffer* sollte auf ein *PTListBoxData*-Objekt zeigen, das als Transferpuffer für die Liste verwendet wird.

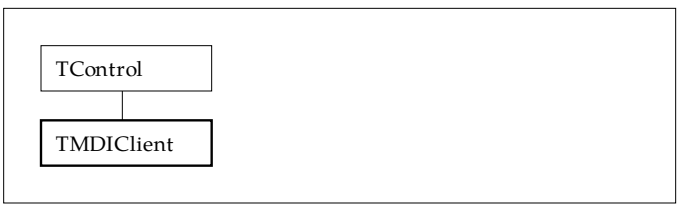
Transfer liefert die Größe von *PTListBoxData* (des Zeigers, nicht der Struktur) zurück. Um die Größe zu ermitteln, ohne daß Daten übertragen werden, übergeben Sie *TF_SIZEDATA* als *TransferFlag*.

Wichtig: Sie müssen in Ihrem Transferpuffer einen Zeiger auf diese Strukturen verwenden. Sie können keine Kopien der Strukturen in Ihren Transferpuffer einbringen und diese Strukturen nicht als Transferpuffer verwenden.

Querverweis: *TListBoxData, TWindowsObject::TransferBuffer*

TMDIClient

mdi.h



Die Client-Fenster der Multidokumentschnittstelle MDI (repräsentiert durch ein *TMDIClient*-Objekt) verwalten die untergeordneten MDI-Fenster eines *TMDIFrame*-Objekts.

Datenelement

ClientAttr

```
LPCLIENTCREATESTRUCT ClientAttr;
```

ClientAttr enthält einen Zeiger auf eine Struktur mit den Attributen für das MDI-Client-Fenster.

Elementfunktionen

Konstruktor

```
TMDIClient(PTMDIFrame AParent,
           PTModule AModule = NULL);
```

Erstellt ein MDI-Client-Fensterobjekt. Ruft einen *TWindow*-Konstruktor auf und übergibt *AParent* und *AModule*. Setzt *Attr.Id* auf den voreingestellten Bezeichner des Client-Fensters, *ID_MDICLIENT*, und *Attr.Style* auf *WS_CHILD | WS_VISIBLE | WS_GROUP | WS_TABSTOP | WS_CLIPCHILDREN*. Diese Funktion initialisiert ferner *ClientAttr*, wobei dessen Element *idFirstChild* auf *ID_FIRSTMDICHILD* gesetzt wird.

Querverweis: *TWindow::TWindow*

Konstruktor

```
TMDIClient(PTMDIFrame AParent, HWND AnHWindow,
           PTModule AModule = NULL);
```

Erstellt ein MDI-Client-Objekt, das mit dem MDI-Client-Fenster verknüpft werden soll, dessen Handle angegeben ist. Die Funktion ruft einen *TWindow*-Konstruktor auf, wobei *AnHWindow* und *AModule* übergeben werden. Dieser Konstruktor kann zur Erzeugung von MDI-Client-Fenstern in dynamischen Linkbibliotheken verwendet werden, in denen das übergeordnete Fenster ein Nicht-ObjectWindows-Fenster ist.

Konstruktor `TMDIClient(StreamableInit);` protected

Stream-Konstruktor von *TMDIClient*. Ruft den Stream-Konstruktor von *TWindow* auf. Er wird aufgerufen, wenn ein *TMDIClient*-Objekt unter Verwendung von Daten aus einem Stream instantiiert wird.

Querverweis: *TWindow::TWindow*

Destruktor `virtual ~TMDIClient();`

Gibt die *ClientAttr*-Struktur frei (falls vorhanden).

Querverweis: *TWindow::~TWindow*

ArrangeIcons `virtual void ArrangeIcons();`

Ordnet die Symbole der untergeordneten MDI-Fenster in einer Zeile im unteren Bereich des MDI-Client-Fensters an.

build `static PTStreamable build();`

Ruft den Konstruktor *TMDIClient* (*StreamableInit*) auf und erstellt ein Objekt vom Typ *TMDIClient*, dessen Datenelemente anschließend aus einem Stream eingelesen werden.

CascadeChildren `virtual void CascadeChildren();`

Vergrößert/verkleinert und ordnet alle untergeordneten MDI-Fenster, die nicht als Symbol dargestellt sind, im MDI-Client-Fenster an. Die untergeordneten Fenster überlappen sich, wobei jedoch alle Titelzeilen sichtbar bleiben.

GetClassName `virtual LPSTR GetClassName();` protected

Liefert "MDICLIENT", den Bezeichner, unter dem die Fensterklasse in Windows registriert wird, zurück.

read `virtual Pvoid read(Ripstream is);` protected

Ruft *TWindow::read* auf, um das Basisobjekt *TWindow* einzulesen. Setzt *ClientAttr->hWindowMenu* auf 0 und liest *ClientAttr->idFirstChild* ein.

TileChildren `virtual void TileChildren();`

Vergrößert/verkleinert und ordnet alle untergeordneten MDI-Fenster, die nicht als Symbol dargestellt sind, im MDI-Client-Fenster an. Die untergeordneten Fenster füllen den gesamten Client-Bereich aus, ohne sich dabei zu überlappen.

WMMDIActivate
```
virtual void WMMDIActivate(RTMessage Msg) =
[WM_FIRST + WM_MDIACTIVATE];
```
protected

Überschreibt *TWindow::WMMDIActivate* und ruft stattdessen *DefWndProc* auf. Falls Sie diese Funktion überschreiben, sollten Sie sie in der neuen Funktion explizit aufrufen, und zwar nach dem von Ihnen geschriebenen Code.

WMPaint `virtual void WMPaint(RTMessage Msg) = [WM_FIRST + WM_PAINT];` protected

Überschreibt *TWindow::WMPaint* , so daß *DefWndProc* aufgerufen wird.

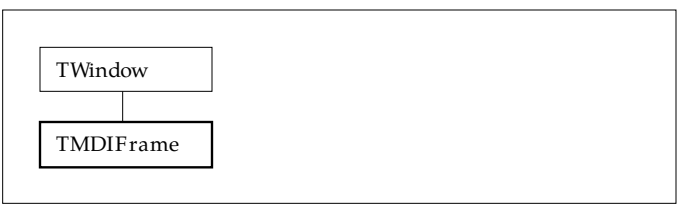
Querverweis: *TWindow::Paint, TScroller::BeginView, TScroller::EndView*

write `virtual void write(Ropstream os);` protected

Ruft *TWindow::write* auf, um zuerst das Basisobjekt *TWindow* und dann *ClientAttr->idFirstChild* an den Stream zu übergeben.

TMDIFrame

mdi.h



Die Rahmenfenster der Multidokumentschnittstelle MDI (repräsentiert von *TMDIFrame*) sind überlappende Fenster, die als Hauptfenster in MDI-Anwendungen dienen. *TMDIFrame*-Objekte verwalten automatisch die in Windows erforderliche Erstellung und Initialisierung eines MDI-Client-»Fensters« (repräsentiert durch ein *TMDIClient*-Objekt). *TMDIFrame* enthält Elementfunktionen, die die untergeordneten MDI-Fenster manipulieren. Dazu gehören die Funktionen *TileChildren* und *CloseChildren*.

Datenelemente

ActiveChild `PTWindow ActiveChild;`

ActiveChild zeigt auf das aktive untergeordnete MDI-Fenster von *TMDIFrame*. *ActiveChild* wird vom untergeordneten Fenster in seiner *WMMDIActivate*-Botschaftsantwortfunktion gesetzt.

ChildMenuPos `int ChildMenuPos;`

ChildMenuPos enthält die Position des Untermenüs der untergeordneten Fenster im Hauptmenü des MDI-Fensters. Die Position Null liegt ganz oben links.

ClientWnd `PTMDIClient ClientWnd;`

ClientWnd zeigt auf das *TMDIFrame*-Client-Fenster, eine Instanz von *TMDIClient*.

Element-funktionen

Konstruktor

```
TMDIFrame(LPSTR ATitle, int MenuId,
          PTModule AModule = NULL);
```

Erstellt ein MDI-Rahmenfensterobjekt und verwendet dabei den Titel *ATitle* und den Integer-Ressourcen-Bezeichner *MenuId*.

Konstruktor

```
TMDIFrame(LPSTR ATitle, LPSTR MenuName,
          PTModule AModule = NULL);
```

Erstellt ein MDI-Rahmenfensterobjekt mit dem Titel *ATitle* und dem Stringressourcen-Bezeichner *MenuName*. MDI-Rahmenfenster müssen über Menüs verfügen und dürfen kein übergeordnetes Fenster haben. *ActiveChild* wird mit *NULL* initialisiert. Per Voreinstellung setzt *TMDIFrame ChildMenuPos* auf Null. Damit wird angezeigt, daß sich das Menü für die untergeordneten Fenster ganz oben links befindet.

Sie können *ChildMenuPos* in dem Konstruktor Ihrer abgeleiteten Klasse neu setzen, z.B.:

```
TMyMDIFrame::TMyMDIFrame(LPSTR ATitle, LPSTR MenuName,
   PTModule AModule = NULL) : TMDIFrame(ATitle, AMenu,
                                        AModule)
{
  ChildMenuPos = 3;
}
```

Konstruktor

```
TMDIFrame(HWND AnHWindow, HWND ClientWnd,
          PTModule AModule = NULL);
```

Erstellt ein MDI-Fensterobjekt, das mit dem MDI-Fenster verknüpft werden soll, dessen Handle angegeben ist. Die Funktion ruft dann einen *TWindow*-Konstruktor auf und erstellt danach ein MDI-Client-Objekt, das mit dem Client-Fenster verknüpft werden soll, dessen Handle angegeben ist. Die Funktion erzeugt also ein Objekt und verknüpft es mit bereits bestehenden Schnitstellenelementen.

Konstruktor

```
TMDIFrame(StreamableInit);                    protected
```

Stream-Konstruktor von *TMDIFrame*. Ruft den Stream-Konstruktor von *TWindow* auf. Er wird aufgerufen, wenn ein *TMDIFrame*-Objekt unter Verwendung von Daten aus einem Stream instantiiert wird.

Querverweis: *TWindow::TWindow*

Destruktor

```
virtual void ~TMDIFrame();
```

Löscht das MDI-Client-Fensterobjekt.

Querverweis: *TWindow::~TWindow*

ArrangeIcons
```
virtual void ArrangeIcons();
```
Ordnet die als Symbol dargestellten, untergeordneten MDI-Fenster in einer Zeile im unteren Bereich des MDI-Client-Fensters an. Dazu wird die Elementfunktion *ArrangeIcons* des Client-Fensters aufgerufen.

Querverweis: *TMDIClient::ArrangeIcons*

build
```
static PTStreamable build();
```
Ruft den Konstruktor *TMDIFrame(StreamableInit)* auf und erstellt ein Objekt vom Typ *TMDIFrame*, dessen Datenelemente anschließend aus einem Stream eingelesen werden.

CascadeChildren
```
virtual void CascadeChildren();
```
Vergrößert/verkleinert und ordnet alle untergeordneten MDI-Fenster, die nicht als Symbol dargestellt sind, im MDI-Client-Fenster an. Dazu wird die Elementfunktion *CascadeChildren* des Client-Fensters aufgerufen.

Querverweis: *TMDIClient::CascadeChildren*

CloseChildren
```
virtual BOOL CloseChildren();
```
Gibt alle untergeordneten MDI-Fenster frei, wenn diese geschlossen werden können. Liefert *TRUE* zurück, wenn alle untergeordneten Fenster geschlossen sind oder wenn es keine untergeordneten Fenster gibt. Wenn ein Fenster nicht geschlossen werden kann, wird *FALSE* zurückgeliefert.

Querverweis: *TWindow::CanClose*

CMArrangeIcons
```
virtual void CMArrangeIcons(RTMessage Msg) =
[CM_FIRST + CM_ARRANGEICONS];                    protected
```
Antwortet mit einem Aufruf von *ArrangeIcons* auf eine Menüauswahl mit der ID *CM_ARRANGEICONS*.

Querverweis: *TMDIFrame::ArrangeIcons*

CMCascadeChildren
```
virtual void CMCascadeChildren(RTMessage Msg) =
[CM_FIRST + CM_CASCADECHILDREN];                 protected
```
Antwortet mit einem Aufruf von *CascadeChildren* auf eine Menüauswahl mit der ID *CM_CASCADECHILDREN*.

Querverweis: *TMDIFrame::CascadeChildren*

CMCloseChildren
```
virtual void CMCloseChildren(RTMessage Msg) =
[CM_FIRST + CM_CLOSECHILDREN];                   protected
```
Antwortet mit einem Aufruf von *CloseChildren* auf eine Menüauswahl mit der ID *CM_CLOSECHILDREN*.

Querverweis: *TMDIFrame::CloseChildren*

CMCreate-Child

```
virtual void CMCreateChild(RTMessage Msg) =
[CM_FIRST + CM_CREATECHILD];                    protected
```

Antwortet mit einem Aufruf von *CreateChild* auf eine Menüauswahl mit der ID *CM_CREATECHILD*, um ein neues untergeordnetes Fenster zu erstellen.

Querverweis: *TMDIFrame::CreateChild*

CMTile-Children

```
virtual void CMTileChildren(RTMessage Msg) =
[CM_FIRST + CM_TILECHILDREN];                   protected
```

Antwortet mit einem Aufruf von *TileChildren* auf eine Menüauswahl mit der ID *CM_TILECHILDREN.*

Querverweis: *TMDIFrame::TileChildren*

CreateChild

```
virtual PTWindowsObject CreateChild();
```

Erstellt und initialisiert durch den Aufruf von *InitChild* und *MakeWindow* ein neues, untergeordnetes MDI-Fenster und liefert einen Zeiger auf das neue Fenster.

Querverweis: *TMDIFrame::InitChild, TModule::MakeWindow*

GetClassName

```
virtual LPSTR GetClassName();                   protected
```

Liefert "OWLMDIFrame", den Bezeichner, unter dem die Fensterklasse in Windows registriert wird, zurück.

GetClient

```
virtual PTMDIClient GetClient();
```

Liefert einen Zeiger auf das MDI-Client-Fenster zurück, der in *ClientWnd* gespeichert ist.

GetWindow-Class

```
virtual void GetWindowClass(WNDCLASS _FAR
                            & AWndClass);       protected
```

Modifiziert die Standard-Fensterklassenstruktur und gibt sie in *AWndClass* zurück. Die Funktion ruft *TWindow::GetWindow-Class* auf und setzt dann *AWndClass.Style* auf 0.

Querverweis: *TWindow::GetWindowClass*

InitChild

```
virtual PWindowsObject InitChild();
```

Erstellt eine Instanz von *TWindow* als untergeordnetes MDI-Fenster und liefert einen Zeiger darauf zurück. Überschreiben Sie diese Elementfunktion in Ihrer abgeleiteten MDI-Fensterklasse, um eine Instanz einer abgeleiteten MDI-Klasse für untergeordnete Fenster zu erstellen, z.B.:

```
PTWindowsObject TMyMDIFrame::InitChild()
{
   return new TMyMDIChild(this, "");
}
```

Querverweis: *TMDIFrame::CreateChild*

InitClient-Window

```
virtual void InitClientWindow();
```

Erstellt das MDI-Client-Fenster als Instanz von *TMDIClient* und speichert dessen Zeiger in *ClientWnd*.

read

```
virtual Pvoid read(Ripstream is);
```

protected

Ruft *TWindow::read* auf, um das Basisobjekt *TWindow* einzulesen. Danach werden *ClientWnd*, *ChildMenuPos* und *ActiveChild* eingelesen.

SetupWindow

```
virtual void SetupWindow();
```

protected

Erstellt mit einem Aufruf von *InitClientWindow* ein MDI-Client-Fensterobjekt. Die Funktion entfernt das MDI-Client-Fenster aus der Liste der untergeordneten Fenster, um es separat zu bearbeiten.

Wenn Sie *SetupWindow* in einer abgeleiteten Klasse überschreiben, müssen Sie sicherstellen, daß *TMDIFrame::SetupWindow* explizit aufgerufen wird.

Querverweis: *TMDIFrame::InitClientWindow*

TileChildren

```
virtual void TileChildren();
```

protected

Vergrößert/verkleinert und ordnet alle untergeordneten MDI-Fenster, die nicht als Symbol dargestellt sind, im MDI-Client-Fenster an. Dazu wird die Elementfunktion *TileChildren* des Client-Fensters aufgerufen.

Querverweis: *TMDIClient::TileChildren*

WMActivate

```
virtual void WMActivate(RTMessage Msg) =
[WM_FIRST + WM_ACTIVATE];
```

protected

Antwortet auf eine eingehende *WM_ACTIVATE*-Botschaft. Da ein untergeordnetes MDI-Fenster keine *WM_MDIACTIVATE*-Botschaften erhält, wenn sein Rahmenfenster aktiviert oder deaktiviert wird, wird hier die Elementfunktion *ActivationResponse* des aktiven untergeordneten Fensters aufgerufen.

Querverweis: *TWindow::ActivationResponse, TWindow::WMActivate*

write

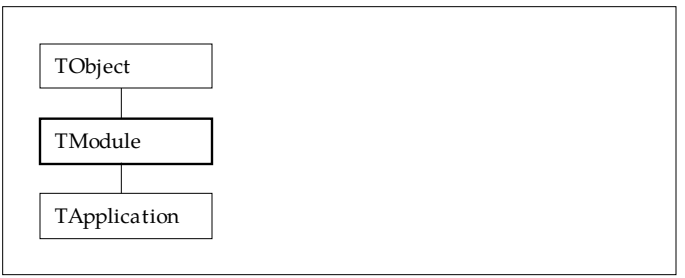
```
virtual void write(Ropstream os);
```

protected

Ruft *TWindow::write* auf, um zuerst das Basisobjekt *TWindow* und danach *ClientWnd*, *ChildMenuPos* und *ActiveChild* an den Stream zu übergeben.

TModule

module.h



Die dynamischen Linkbibliotheken von ObjectWindows erstellen eine Instanz von *TModule*, die als objektorientierter Repräsentant für das Bibliotheksmodul fungiert.

ObjectWindows-Anwendungen erstellen eine Instanz von *TApplication*, einer von *TModule* abgeleitete Klasse. *TModule* definiert das Verhalten und die Eigenschaften, die sowohl für das Bibliotheks- als auch das Anwendungsmodul zutreffen.

Die *TModule*-Elementfunktionen unterstützen die Fenster-Speicherverwaltung und die Fehlerbehandlung.

Datenelemente

hInstance `HANDLE hInstance;`

Enthält das Handle auf die ausführende Instanz einer Windows-Anwendung oder eines DDL-Moduls. Das Handle muß Windows als Parameter zur Verfügung stehen, wenn die Ressourcen geladen werden.

lpCmdLine `LPSTR lpCmdLine;`

Zeigt auf eine Kopie der Kommandozeile, die beim Laden des Moduls übergeben wurde. Es handelt sich dabei um einen null-terminierten String.

Name `LPSTR Name;`

Name speichert den Namen des Anwendungsmoduls. Dieses Element kann intern vom Modul-Code verwendet werden.

Status `int Status;`

Status speichert den Status des Moduls. Es kann von ObjectWindows gesetzt werden, um einen Fehler bei der Erstellung des Hauptfensters für ein Anwendungsmodul anzuzeigen (als Rückgabewert von *WinMain*). *Status* kann aber auch gesetzt werden, um beim Start eines Bibliotheksmoduls (DLL) einen Fehler anzugeben. ObjectWindows verwendet *Status*, um den Wert festzulegen, den *LibMain* zurückgibt.

Element-funktionen

Konstruktor

```
TModule(LPSTR AName, HANDLE AnInstance,
        LPSTR ACmdLine);
```

Initialisiert die Datenelemente mit den angegebenen Parametern. Erstellt eine Kopie von *ACmdLine* und setzt *lpCmdLine* auf diese Kopie. Setzt *Status* auf *EM_INVALIDMODULE*, wenn ein Fehler auftritt.

Destruktor

```
virtual ~TModule();
```

Destruktor für ein *TModule*-Objekt; löscht *lpCmdLine*.

Error

```
virtual void Error(int ErrorCode);
```

Error bearbeitet die Fehler, die von dem in *ErrorCode* angegebenen Fehlerwert gekennzeichnet werden. *ErrorCode* kann ein von Ihnen definierter Fehler oder eine der *EM_*-Konstanten sein, die in Kapitel 24 »Weitere Komponenten von ObjectWindows« beschrieben werden.

Error zeigt den Fehlercode in einem Meldungsfenster an und fragt den Anwender, ob weitergearbeitet werden soll. Falls das Programm nicht fortgesetzt werden soll, wird es beendet. Wenn der Anwender weiterarbeitet, ist es nicht sicher, ob das Programm noch regulär fortgesetzt werden kann.

ExecDialog

```
virtual int ExecDialog(PTWindowsObject ADialog);
```

Führt einen modalen Dialog aus, wenn das angegebene Dialogobjekt gültig ist. *ExecDialog* ruft *ValidWindow* auf, um die Gültigkeit des Objekts feststellen zu lassen. Wenn das Objekt gültig ist, wird die Elementfunktion *Execute* des Dialogs aufgerufen und das Ergebnis des Aufrufs zurückgeliefert.Wenn das Objekt ungültig ist, liefert die Funktion *IDCANCEL* zurück. Ungültige Objekte und Objekte, deren Ausführung nicht erfolgreich war (*Status* ungleich Null), werden von *ExecDialog* gelöscht, andernfalls von *Execute*.

Querverweis: *TModule::ValidWindow, TDialog::Execute, TModule::MakeWindow*

GetClient-Handle

```
HWND GetClientHandle(HWND AnHWindow);
```

Liefert das Handle eines MDI-Client-Fensters für das Fenster zurück, dessen Handle angegeben ist. Die Funktion liefert 0 zurück, wenn das Fenster kein MDI-Client-Fenster hat.

GetParent-Object

```
virtual PTWindowsObject GetParentObject
                                    (HWND ParentHandle);
```

Diese Funktion wird in einer DLL aufgerufen, um einen Zeiger auf ein übergeordnetes Fensterobjekt zu erhalten. Wenn der Aufrufer Teil einer ObjectWindows-Anwendung ist, liefert *Get-*

ParentObject einen Zeiger auf das übergeordnete Fensterobjekt im Aufrufer zurück.

Bei einem Nicht-ObjectWindows-Aufrufer liefert *GetParentObject* einen Zeiger auf ein Objekt zurück, das in der DLL als Objekt für das übergeordnete Fenster in der aufrufenden Anwendung erstellt ist.

hashValue

```
virtual hashValueType hashValue() const;
```

Für Container-Klassen erforderlich. Überschreibt die abstrakte Funktion in der Klasse *Object*. Diese Funktion liefert *hInstance*, den Hash-Wert eines *TModule*, zurück.

isA

```
virtual classType isA() const;
```

Für Container-Klassen erforderlich. Überschreibt die abstrakte Funktion in der Klasse *Object*. Diese Funktion liefert *moduleClass*, den Klassenbezeichner eines *TModule*, zurück.

isEqual

```
virtual int isEqual(RCObject module) const;
```

Für Container-Klassen erforderlich. Überschreibt die abstrakte Funktion in der Klasse *Object*. Diese Funktion liefert *TRUE* zurück, wenn this und *module* dieselben *hInstance*-Werte haben.

LowMemory

```
BOOL LowMemory();
```

Liefert *TRUE* zurück, wenn der Sicherheitsbereich ausgeschöpft ist.

Querverweis: *TModule::RestoreMemory*

MakeWindow

```
virtual PTWindowsObject MakeWindow(PTWindowsObject
                                   AWindowsObject);
```

Erstellt ein Fenster oder ein nicht-modales Dialogelement, das mit dem angegebenen Objekt verknüpft werden soll, sofern das Objekt gültig ist. *MakeWindow* ruft *ValidWindow* auf, um die Gültigkeit des Objekts festzustellen. Wenn das Objekt gültig ist, wird die Elementfunktion *Create* des Objekts aufgerufen. Wenn *Create TRUE* zurückliefert, wird *AWindowsObject* zurückgegeben.

Wichtig: Beachten Sie, daß *MakeWindow* ungültige Objekte und Objekte, die nicht erstellt werden können, löscht und dabei *NULL* zurückliefert.

Querverweis: *TModule::ValidWindow, TWindow::Create*

nameOf

```
virtual Pchar nameOf() const;
```

Für Container-Klassen erforderlich. Überschreibt die abstrakte Funktion in der Klasse *Object*.

Diese Funktion liefert "TModule", den Klassenbezeichnungsstring eines *TModule*-Objekts, zurück.

printOn `virtual void printOn(Rostream outputStream) const;`

Für Container-Klassen erforderlich. Überschreibt die abstrakte Funktion in der Klasse *Object*. Diese Funktion gibt eine hexadezimale Darstellung des Instanzen-Handles für das Modul (*hInstance*) in den angegebenen Stream aus.

RestoreMemory `void RestoreMemory();`

Reserviert Speicherplatz für einen Sicherheitsbereich (wenn dies noch nicht geschehen ist).

Querverweis: *SafetyPool::Allocate, TModule::LowMemory*

ValidWindow `virtual PTWindowsObject ValidWindow(PTWindowsObject AWindowsObject);`

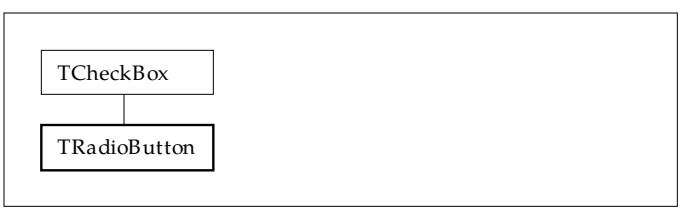
Stellt fest, ob *AWindowsObject* ein gültiges Objekt ist. Wenn *AWindowsObject* gültig ist, liefert *ValidWindow* einen Zeiger darauf zurück, andernfalls wird das Objekt gelöscht und *NULL* zurückgeliefert.

AWindowsObject ist ungültig, wenn eine der beiden folgenden Bedingungen gegeben ist: wenn *LowMemory TRUE* zurückliefert (eine Speicherplatzreservierung für das Objekt gefährdet den Sicherheitsbereich) oder wenn das Datenelement *Status* von *AWindowsObject* ungleich Null ist.

Querverweis: *TModule::LowMemory, TWindowsObject::Status*

TRadioButton

radiobut.h



TRadioButton ist ein Schnittstellenobjekt, das ein entsprechendes Schaltfeld in Windows repräsentiert. *TRadioButton* wird für die Erstellung eines Schaltfeld-Steuerelements in einem übergeordneten *TWindow*-Objekt verwendet. *TRadioButton* vereinfacht auch die Kommunikation zwischen Ihrer Anwendung und den Schaltfeld-Steuerelementen eines *TDialog*-Objektes.

Schaltfelder haben zwei »Zustände«: eingeschaltet und ausgeschaltet. *TRadioButton* erbt Elementfunktionen für die Status-Verwaltung von der Basisklasse *TCheckBox*. Ein Schaltfeld kann auch Teil einer Gruppe (*TGroupBox*) sein, die ihre Steuerelemente optisch und funktional zusammenfaßt.

Element-funktionen

Konstruktor

```
TRadioButton(PTWindowsObject AParent, int AnId,
             LPSTR ATitle, int X, int Y, int W, int H,
             PTGroupBox AGroup,
             PTModule AModule = NULL);
```

Erstellt ein Schaltfeldobjekt mit dem angegebenen übergeordneten Fenster (*AParent*), mit der Steuerelement-ID (*AnId*), mit dem zugehörigen Text (*ATitle*), mit der Position (*X*, *Y*) relativ zum Ursprung des Client-Bereichs des übergeordneten Fensters, mit der Breite (*W*), der Höhe (*H*) und der zugehörigen Gruppe (*AGroup*).

Die Funktion ruft den *TCheckBox*-Konstruktor mit den entsprechenden Parametern auf und setzt das Datenelement *Attr.Style* auf *WS_CHILD | WS_VISIBLE | BS_AUTORADIOBUTTON*.

Querverweis: *TControl::TControl*

Konstruktor

```
TRadioButton(PTWindowsObject AParent, int ResourceId,
             PTGroupBox AGroup, PTModule AModule = NULL);
```

Erstellt ein *TRadioButton*-Objekt, das mit einem Schaltfeld-Steuerelement eines *TDialog*-Objekts verknüpft werden soll, und ruft den *TCheckBox*-Konstruktor mit identischen Parametern auf. Der Parameter *ResourceId* muß einer Schaltfeldressource entsprechen, die Sie definieren.

Konstruktor

```
TRadioButton(StreamableInit);
```

protected

Stream-Konstruktor von *TRadioButton*. Ruft den Stream-Konstruktor von *TCheckBox* auf. Er wird aufgerufen, wenn ein *TRadioButton*-Objekt unter Verwendung von Daten aus einem Stream instantiiert wird.

Querverweis: *TCheckBox::TCheckBox*

BNClicked

```
virtual void BNClicked(RTMessage Msg) =
[NF_FIRST + BN_CLICKED];
```

Beantwortet die Windows-Botschaft *BN_CLICKED* durch Aufruf von *TCheckBox::BNClicked*.

build

```
static PTStreamable build();
```

Ruft den Konstruktor *TRadioButton*(*StreamableInit*) auf und erstellt ein Objekt vom Typ *TRadioButton*, dessen Datenelemente anschließ end aus einem Stream eingelesen werden.

TScrollBar

scrollba.h

TScrollBar-Objekte repräsentieren selbständige vertikale und horizontale Bildlaufleisten. Die meisten *TScrollBar*-Elementfunktionen betreffen die Verwaltung von Position und Bereich des Positionszeigers.Eine spezielle Eigenschaft von *TScrollBar* sind die auf Benachrichtigungen basierenden Elementfunktionen, die die Position des Positionszeigers automatisch als Antwort auf Bildlaufleisten-Botschaften von Windows anpassen.

Wichtig: *TScrollBar*-Objekte dürfen niemals in Fenster plaziert werden, die unter ihren Attributen die Stile *WS_HSCROLL* oder *WS_VSCROLL* haben.

Datenelemente

LineMagnitude `int LineMagnitude;`

LineMagnitude ist die Größe der Schrittweite, um die der Bildlauf durchgeführt werden soll, wenn der Anwender die Pfeilsymbole der Bildlaufleisten anklickt. Der *TScrollBar*-Konstruktor setzt *LineMagnitude* per Voreinstellung auf 1. Der Bildlaufbereich liegt per Voreinstellung zwischen 0 und 100.

Querverweis: *TScrollBar::SetupWindow*

PageMagnitude `int PageMagnitude;`

PageMagnitude ist die Größe der Schrittweite, um die der Bildlauf durchgeführt werden soll, wenn der Anwender durch Anklicken des Bildlaufbereichs der Bildlaufleiste eine größere Verschiebung anfordert. Der *TScrollBar*-Konstruktor setzt *PageMagnitude* per Voreinstellung auf 10. Der Bildlaufbereich liegt standardmäßig zwischen 0 und 100.

Elementfunktionen

Konstruktor

```
TScrollBar(PTWindowsObject AParent, int AnId, int X,
           int Y, int W, int H, BOOL IsHScrollBar,
           PTModule AModule = NULL);
```

Erstellt und initialisiert ein *TScrollBar*-Objekt mit dem gegebenen übergeordneten Fenster (*AParent*), der Steuerelement-ID (*AnId*), mit der Position (*X*, *Y*), mit der Breite (*W*) und der Höhe

(*H*). Die Funktion ruft den *TControl*-Konstruktor mit den entsprechenden Parametern auf. Wenn *IsHScrollBar TRUE* ist, fügt sie zu den in *Attr.Style* angegebenen Stilen *SBS_HORZ* hinzu. Ist *IsHScrollBar FALSE*, wird *SBS_VERT* eingefügt. Wenn die angegebene Höhe für eine horizontale Bildlaufleiste oder die angegebene Breite für eine vertikale Bildlaufleiste Null beträgt, wird der Standardwert verwendet. *LineMagnitude* wird mit 1 und *PageMagnitude* wird mit 10 initialisiert.

Querverweis: *TControl::TControl*

Konstruktor

```
TScrollBar(PTWindowsObject AParent, int ResourceId,
           PTModule AModule = NULL);
```

Erstellt ein *TScrollBar*-Objekt, das mit einem Bildlaufleisten-Steuerelement eines *TDialog*-Objekts verknüpft werden soll, und ruft den *TControl*-Konstruktor mit identischen Parametern auf.

Der Parameter *ResourceId* muß einer Bildlaufleistenressource entsprechen, die Sie definieren.

Konstruktor

```
TScrollBar(StreamableInit);
```

protected

Stream-Konstruktor von *TScrollBar*. Ruft den Stream-Konstruktor von *TControl* auf. Er wird aufgerufen, wenn ein *TScrollBar*-Objekt unter Verwendung von Daten aus einem Stream instantiiert wird.

Querverweis: *TControl::TControl*

build

```
static PTStreamable build();
```

Ruft den Konstruktor *TScrollBar*(*StreamableInit*) auf und erstellt ein Objekt vom Typ *TScrollBar*, dessen Datenelemente anschließend aus einem Stream eingelesen werden.

DeltaPos

```
int DeltaPos(int Delta);
```

Ruft *SetPosition* auf, um die Position des Positionszeigers um den in *Delta* angegebenen Wert zu ändern. Ein positiver Wert in *Delta* verschiebt den Positionszeiger nach unten oder rechts, ein negativer Wert nach oben oder links.

Die neue Position wird zurückgeliefert.

Querverweis: *TScrollBar::SetPosition*

GetClassName

```
virtual LPSTR GetClassName();
```

protected

Liefert "SCROLLBAR", den Bezeichner, unter dem die Fensterklasse in Windows registriert wird, zurück.

GetPosition

```
int GetPosition();
```

Liefert die aktuelle Position des Positionszeigers zurück.

Querverweis: *TScrollBar::SetPosition*

GetRange
```
void GetRange(Rint LoVal, Rint HiVal);
```
Liefert in *LoVal* und *HiVal* die Grenzen des aktuellen Bildlaufbereichs zurück.

Querverweis: *TScrollBar::SetPosition, TScrollBar::SetRange*

read
```
virtual Pvoid read(Ripstream is);
```
protected

Ruft *TWindow::read* auf, um das Basisobjekt *TWindow* einzulesen. Danach werden *LineMagnitude* und *PageMagnitude* eingelesen.

SBBottom
```
virtual void SBBottom(RTMessage Msg) =
[NF_FIRST + SB_BOTTOM];
```
protected

Bewegt den Positionszeiger mit einem Aufruf von *SetPosition* an das untere Ende oder auf die rechte Seite der Bildlaufleiste. Diese Elementfunktion wird aufgerufen, wenn der Positionszeiger auf die unterste oder die am weitesten rechts liegende Position der Bildlaufleiste gezogen wird.

Querverweis: *TScrollBar::SetPosition*

SBLineDown
```
virtual void SBLineDown(RTMessage Msg) =
[NF_FIRST + SB_LINEDOWN];
```
protected

Bewegt den Positionszeiger durch Aufruf von *SetPosition* um die in *LineMagnitude* angegebene Schrittweite nach unten oder rechts. Diese Elementfunktion wird als Antwort auf das Anklicken des unteren oder rechten Pfeils der Bildlaufleiste aufgerufen.

Querverweis: *TScrollBar::SetPosition*

SBLineUp
```
virtual void SBLineUp(RTMessage Msg) =
[NF_FIRST + SB_LINEUP];
```
protected

Bewegt den Positionszeiger durch Aufruf von *SetPosition* um die in *LineMagnitude* angegebene Schrittweite nach oben oder links. Diese Elementfunktion wird als Antwort auf das Anklicken des oberen oder linken Pfeils der Bildlaufleiste aufgerufen.

Querverweis: *TScrollBar::SetPosition*

SBPageDown
```
virtual void SBPageDown(RTMessage Msg) =
[NF_FIRST + SB_PAGEDOWN];
```
protected

Bewegt den Positionszeiger durch Aufruf von *SetPosition* um die in *PageMagnitude* angegebene Schrittweite nach unten oder rechts. Diese Elementfunktion wird als Antwort auf das Anklicken des unteren oder rechten Bildlaufbereichs der Bildlaufleiste aufgerufen.

Querverweis: *TScrollBar::SetPosition*

SBPageUp
```
virtual void SBPageUp(RTMessage Msg) =
[NF_FIRST + SB_PAGEUP];                    protected
```

Bewegt den Positionszeiger durch Aufruf von *SetPosition* um die in *PageMagnitude* angegebene Schrittweite nach oben oder links. Diese Elementfunktion wird als Antwort auf das Anklicken des oberen oder linken Bildlaufbereichs der Bildlaufleiste aufgerufen.

Querverweis: *TScrollBar::SetPosition*

SBThumb-Position
```
virtual void SBThumbPosition(RTMessage Msg) =
[NF_FIRST + SB_THUMBPOSITION];             protected
```

Verschiebt den Positionszeiger durch Aufruf von *SetPosition*. Diese Elementfunktion wird aufgerufen, wenn der Positionszeiger an eine neue Position gebracht werden soll.

Querverweis: *TScrollBar::SetPosition*

SBThumbTrack
```
virtual void SBThumbTrack(RTMessage Msg) =
[NF_FIRST + SB_THUMBTRACK];                protected
```

Verschiebt den Positionszeiger durch Aufruf von *SetPosition* an eine neue Position, wenn er gezogen wird.

Querverweis: *TScrollBar::SetPosition*

SBTop
```
virtual void SBTop(RTMessage Msg) =
[NF_FIRST + SB_TOP];                       protected
```

Verschiebt den Positionszeiger durch Aufruf von *SetPosition* an den Beginn oder auf die rechte Seite der Bildlaufleiste. Diese Elementfunktion wird aufgerufen, wenn der Positionszeiger auf die Anfangsposition oder die am weitesten rechts gelegene Position der Bildlaufleiste gezogen wird.

Querverweis: *TScrollBar::SetPosition*

SetPosition
```
void SetPosition(int ThumbPos);
```

Verschiebt den Positionszeiger an die in *ThumbPos* angegebene Position. Wenn *ThumbPos* außerhalb des aktuellen Bildlaufbereichs liegt, wird der Positionszeiger an die dazu nächstgelegene Position innerhalb des Bereichs gebracht.

Querverweis: *TScrollBar::GetPosition*

SetRange
```
void SetRange(int LoVal, int HiVal);
```

Setzt die Grenzen des Bildlaufbereichs auf *LoVal* und *HiVal*.

Querverweis: *TScrollBar::GetRange*

SetupWindow
```
virtual void SetupWindow();                protected
```

Setzt den Bildlaufbereich auf 0 bis 100. Mit *SetRange* kann dieser Bereich neu definiert werden.

Querverweis: *TScrollBar::SetRange*

Transfer `virtual WORD Transfer(Pvoid DataPtr, WORD TransferFlag);`

Überträgt Bildlaufleisten-Daten (die Grenzen des Bildlaufbereichs und den Positionszeiger) in oder aus dem Transferpuffer, auf den *DataPtr* zeigt. *DataPtr* sollte auf eine *TScrollBarData*-Struktur zeigen (vgl. dazu Kapitel 24 »Weitere Komponenten von ObjectWindows«).

Daten werden in oder aus dem Transferpuffer übertragen, wenn *TransferFlag* auf *TF_GETDATA* oder *TF_SETDATA* gesetzt ist.

Transfer liefert immer den Umfang der übertragenen Daten (die Größe der *TScrollBarData*-Struktur) zurück. Wenn die Größe dieser Daten ermittelt werden soll, ohne daß Daten übertragen werden, muß *TransferFlag* auf *TF_SIZEDATA* gesetzt werden.

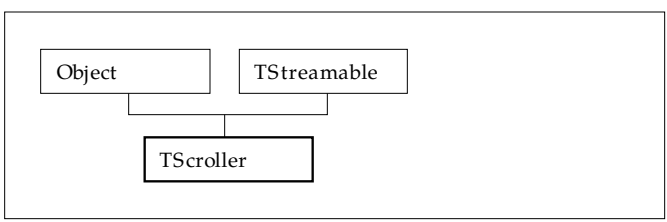
write `virtual void write(Ropstream os);` protected

Ruft *TWindow::write* auf, um zuerst das Basisobjekt *TWindow* in den Stream zu schreiben.

Danach werden *LineMagnitude* und *PageMagnitude* in den Stream geschrieben.

TScroller

scroller.h



TScroller-Instanzen unterstützen den Bildlauf eines *TWindow*-Objekts – normalerweise zusammen mit den Bildlaufleisten des Fensters. Andererseits wird ein automatischer Bildlauf durchgeführt, wenn die Maus aus dem Fenster hinausgezogen wird.

Um diese *TSrcoller*-Eigenschaft verwenden zu können, wird das Datenelement *Scroller* Ihrer von *TWindow* abgeleiteten Klasse auf ein *TScroller*-Objekt gesetzt, das im *TWindow*-Konstruktor dieser abgeleiteten Klasse instantiiert wird.

Datenelemente

AutoMode `BOOL AutoMode;`

AutoMode ist *TRUE*, wenn der automatische Bildlauf für *TScroller* aktiviert ist. Wenn *AutoMode TRUE* ist, wird für das zugehö-

rige Fenster ein Bildlauf durchgeführt, wenn die Maus aus dem Fenster hinausgezogen wird. *AutoMode* ist per Voreinstellung *TRUE*.

AutoOrg `BOOL AutoOrg;`

AutoOrg ist ein Flag (per Voreinstellung auf *TRUE* gesetzt), das für das Zeichnen des Fensters anzeigt, daß der Scroller den Ursprungspunkt des Client-Bereichs automatisch um *XPos* und *YPos* aktualisiert.

Querverweis: *TScroller::BeginView*

ClassHashValue `static hashValueType ClassHashValue;` protected

Enthält den Hash-Wert der zuletzt erstellten Instanz von *TScroller*. Dieses Element wird als eindeutiger Schlüssel für *TScroller*-Instanzen eingesetzt.

HasHScrollBar `BOOL HasHScrollBar;`

Wenn das zugehörige Fenster eine horizontale Bildlaufleiste hat, ist *HasHScrollBar TRUE*.

HasVScrollBar `BOOL HasVScrollBar;`

Wenn das zugehörige Fenster eine vertikale Bildlaufleiste hat, ist *HasVScrollBar TRUE*.

InstanceHash-Value `hashValueType InstanceHashValue;` protected

Enthält den Hash-Wert von this und stellt den eindeutigen Schlüssel für *hashValue* dar.

TrackMode `BOOL TrackMode;`

TrackMode ist *TRUE*, wenn für das zugehörige Fenster der Bildlauf durchgeführt werden soll, während die Maus gezogen wird (»Nachführung«). Wenn *TrackMode FALSE* ist, wird der Bildlauf erst durchgeführt, wenn das Ziehen des Positionszeigers beendet ist. *TrackMode* ist per Voreinstellung *TRUE*.

Window `PTWindow Window;`

Window zeigt auf das übergeordnete Fenster von *TScroller*.

XLine `int XLine;`

XLine ist die Anzahl der *XUnit*-Schritte, um die das Fensterinnere horizontal verschoben werden soll, wenn der Pfeil der horizontalen Bildlaufleiste angeklickt wird.

XPage `int XPage;`

XPage ist die Anzahl der *XUnit*-Schritte, um die der Fensterinhalt verschoben werden soll, wenn der horizontale Bildlaufbereich angeklickt wird. Per Voreinstellung ist *XPage* gleich der

aktuellen Breite des Fensters in *XUnit*-Schritten. Bei einer Änderung der Fenstergröße wird dieser Wert aktualisiert.

Querverweis: *TSCroller::SetPageSize*

XPos `long XPos;`

XPos ist die aktuelle horizontale Position des Scrollers, angegeben in *XUnit*-Schritten.

XRange `long XRange;`

XRange ist die Gesamtanzahl der horizontalen *XUnit*-Schritte, um die ein Bildlauf durchgeführt werden kann. *XRange*-Werte werden an den Konstruktor übergeben, können aber später mit einem Aufruf von *SetRange* geändert werden.

XUnit `int XUnit;`

XUnit ist die virtuelle, horizontale Schrittweite (ausgedrückt in Geräteeinheiten), die von *TScroller* verwendet wird. Der *XUnit*-Wert wird an den Konstruktor übergeben, kann aber später mit einem Aufruf von *SetUnits* geändert werden.

YLine `int YLine;`

YLine ist die Anzahl der *YUnit*-Schritte, um die als Antwort auf das Anklicken des Pfeils der vertikalen Bildlaufleiste vertikal geblättert werden soll.

YPage `int YPage;`

YPage ist die Anzahl der *YUnit*-Schritte, um die der Fensterinhalt verschoben werden soll, wenn der vertikale Bildlaufbereich angeklickt wird. Per Voreinstellung ist *YPage* gleich der aktuellen Höhe des Fensters in *YUnit*-Schritten. Bei einer Änderung der Fenstergröße wird dieser Wert aktualisiert.

Querverweis: *TScroller::SetPageSize*

YPos `long YPos;`

YPos ist die aktuelle vertikale Position des Scrollers, angegeben in *YUnit*-Schritten.

YRange `long YRange;`

YRange ist die Gesamtanzahl der vertikalen *YUnit*-Schritte, um die ein Bildlauf durchgeführt werden kann. *YRange*-Werte werden an den Konstruktor übergeben, können aber später mit einem Aufruf von *SetRange* geändert werden.

YUnit `int YUnit;`

YUnit ist die virtuelle, vertikale Schrittweite (ausgedrückt in Geräteeinheiten), die von *TScroller* verwendet wird. Der *YUnit*-Wert wird an den Konstruktor übergeben, kann aber später mit einem Aufruf von *SetUnits* geändert werden.

Element-funktionen

Konstruktor
```
TScroller(PTWindow TheWindow, int TheXUnint, TheYUnit,
          long TheXRange, long TheYRange);
```

Erstellt ein *TScroller*-Objekt mit *TheWindow* als zugehörigem Fenster, und *TheXUnit*, *TheYUnit*, *TheXRange* und *TheYRange* als *XUnit*, *YUnit*, *XRange* und *YRange*. Initialisiert Datenelemente mit den voreingestellten Werten. *HasHScrollBar* und *HasVScrollBar* werden entsprechend den Bildlaufleisten-Attributen des zugehörigen Fensters gesetzt.

Konstruktor
```
TScroller(StreamableInit);                    protected
```

Stream-Konstruktor von *TScroller*. Er wird aufgerufen, wenn ein *TScroller*-Objekt unter Verwendung von Daten aus einem Stream instantiiert wird.

Destruktor
```
~TScroller();
```

Löscht das *TScroller*-Objekt. Setzt *Scroller* auf *NULL*.

AutoScroll
```
virtual void AutoScroll();
```

Führt für das zugehörige Fenster einen Bildlauf durch, wenn die Maus aus dem Fenster gezogen wird. Die Richtung und der Umfang des Bildlaufs hängen von der aktuellen Position der Maus ab.

BeginView
```
virtual void BeginView(HDC PaintDC, PAINTSTRUCT _FAR &
                       PaintInfo);
```

Wenn *AutoOrg TRUE* ist (Standardbedingung), wird während einer Zeichnen-Operation der Ursprung des Koordinatensystems des Client-Bereichs automatisch um *XPos* und *YPos* aktualisiert.

Wenn *AutoOrg FALSE* ist (z.B. wenn der Scroller größer als 32767 Einheiten ist), müssen Sie den Ursprung selbst aktualisieren.

build
```
static PTStreamable build();
```

Ruft den Konstruktor *TScroller*(*StreamableInit*) auf und erstellt ein Objekt vom Typ *TScroller*, dessen Datenelemente anschliessend aus einem Stream eingelesen werden.

EndView
```
virtual void EndView();
```

Aktualisiert die Positionszeiger der Bildlaufleisten im zugehörigen Fenster so, daß sie der Position von *TScroller* entsprechen.

hashValue
```
virtual hashValueType hashValue() const;
```

Für Container-Klassen erforderlich. Überschreibt die abstrakte Funktion in der Klasse *Object*. Liefert *InstanceHashValue* zurück, den eindeutigen Bezeichner für eine *TScroller*-Instanz.

HScroll `virtual void HScroll(WORD ScrollEvent, int ThumbPos);`

Reagiert mit dem Aufruf von *ScrollBy* und *ScrollTo* auf das gegebene horizontale Bildlaufereignis (*ScrollEvent*). Der Typ des Bildlaufereignisses wird durch die entsprechenden Windows-*SB_*-Konstanten (vgl. Kapitel 24 »Weitere Komponenten von ObjectWindows«) gekennzeichnet. *ThumbPos* speichert die aktuelle Position des Positionszeigers, wenn der Scroller von den Bildlaufereignissen *SB_THUMBTRACK* und *SB_THUMBPOSITION* benachrichtigt wird.

Querverweis: *TWindow::WMHScroll*

isA `virtual classType isA() const;`

Für Container-Klassen erforderlich. Überschreibt die abstrakte Funktion in der Klasse *Object*. Diese Funktion liefert *scrollerClass*, den Klassenbezeichner von *TScroller*, zurück.

isEqual `virtual int isEqual(RCObject testobj) const;`

Für Container-Klassen erforderlich. Überschreibt die abstrakte Funktion in der Klasse *Object*. Diese Funktion liefert *TRUE* zurück, wenn this auf das angegebene *testobj* zeigt.

isVisibleRect `BOOL IsVisibleRect(long X, long Y, int XExt, int YExt);`

Liefert *TRUE* zurück, wenn ein Teil des angegebenen Rechtecks im zugehörigen Fenster sichtbar ist.

nameOf `virtual Pchar nameOf() const;`

Für Container-Klassen erforderlich. Überschreibt die abstrakte Funktion in der Klasse *Object*. Diese Funktion liefert "TScroller", den Klassenenbezeichnungsstring *TScroller*, zurück.

printOn `virtual void printOn(Rostream outputStream) const;`

Für Container-Klassen erforderlich. Überschreibt die abstrakte Funktion in der Klasse *Object*. Diese Funktion gibt eine hexadezimale Darstellung von *Window* in den angegebenen Stream aus.

read `virtual Pvoid read(Ripstream is);` protected

Liest *AutoMode, AutoOrg, HasHScrollBar, HasVScrollbar, TrackMode, XLine, XPage, XPos, XRange, XUnit, YLine, YPage, YPos, YRange* und *YUnit* ein.

Querverweis: *TScroller::write*

ScrollBy `void ScrollBy(long Dx, long Dy);`

Führt durch einen Aufruf von *ScrollTo* für das zugehörige Fenster einen Bildlauf um *Dx* und *Dy* durch.

Querverweis: *TScroller::ScrollTo*

ScrollTo `virtual void ScrollTo(long X, long Y);`

Bewegt den Positionszeiger zu den angegebenen Koordinaten oder an die Grenzen des Bildlaufbereichs, falls die Koordinaten außerhalb liegen, und bewirkt, daß eine *WMPaint*-Botschaft gesendet wird. Wenn ein Teil des Client-Bereichs sichtbar bleibt, wird der Bildlauf zuerst für den Inhalt des Client-Bereichs durchgeführt.

Querverweis: *TWindow::WMPaint, TWindow::Paint*

SetPageSize `virtual void SetPageSize();`

Setzt die Datenelemente *XPage* und *YPage* auf die Breite und Höhe (in *XUnit*-Schritte und *YUnit*-Schritte) des Client-Bereichs im zugehörigen Fenster.

Querverweis: *TWindow::WMSize*

SetRange `void SetRange(long TheXRange, long TheYRange);`

Setzt *XRange* und *YRange* von *TScroller* auf die angegebenen Parameter. Danach ruft die Funktion *SetSBarRange* auf, um die Bildlaufbereiche für die Bildlaufleisten des zugehörigen Fensters zu synchronisieren.

Querverweis: *TScroller::SetSBarRange*

SetSBarRange `virtual void SetSBarRange();`

Setzt die Bildlaufbereiche für die Bildlaufleisten des zugehörigen Fensters so, daß sie mit dem Bereich von *TScroller* übereinstimmen.

SetUnits `void SetUnits(int TheXUnit, int TheYUnit);`

Setzt die Datenelemente *XUnit* und *YUnit* auf *TheXUnit* und *TheYUnit* und aktualisiert *XPage* und *YPage* mit einem Aufruf von *SetPageSize*.

VScroll `virtual void VScroll(WORD ScrollEvent, int ThumbPos);`

Antwortet mit dem Aufruf von *ScrollBy* und *ScrollTo* auf das angegebene vertikale Bildlaufereignis (*ScrollEvent*). Der Typ des Bildlaufereignisses wird durch die entsprechenden Windows-*SB_*-Konstanten (vgl. Kapitel 4 »Windows-Stile und -Konstanten« im *Windows-Referenzhandbuch* «) gekennzeichnet. *ThumbPos* speichert die aktuelle Position des Positionszeigers, wenn der Scroller von den Bildlaufereignissen *SB_THUMBTRACK* oder *SB_THUMBPOSITION* benachrichtigt wird.

Querverweis: *TScroller::ScrollTo, TWindow::WMVScroll*

write `virtual void write(Ropstream os);` protected

Übergibt *AutoMode, AutoOrg, HasHScrollBar, HasVScrollbar, TrackMode, XLine, XPage, XPos, XRange, XUnit, YLine, YPage, YPos, YRange* und *YUnit* an den Stream.

XRangeValue `long XRangeValue(int AScrollUnit);`

Konvertiert einen Bildlaufwert der horizontalen Bildlaufleiste in einen Bereichswert.

XScrollValue `int XScrollValue(long ARangeUnit);`

Konvertiert einen Bereichswert der horizontalen Bildlaufleiste in einen Bildlaufwert.

YRangeValue `long YRangeValue(int AScrollUnit);`

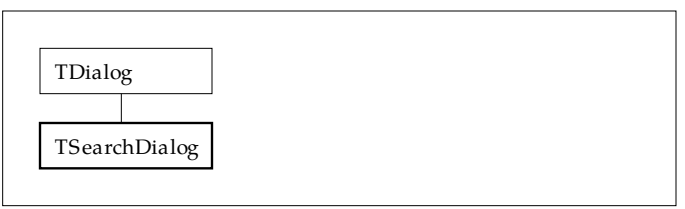
Konvertiert einen Bildlaufwert der vertikalen Bildlaufleiste in einen Bereichswert.

YScrollValue `int YScrollValue(long ARangeUnit);`

Konvertiert einen Bereichswert der vertikalen Bildlaufleiste in einen Bildlaufwert.

TSearchDialog

editwnd.h



TSearchDialog sucht und ersetzt vom Anwender eingegebenen Text unter Verwendung der angegebenen Optionen. Mit Hilfe einer *TSearchStruct*-Struktur, die an den Konstruktor übergeben wird, werden die Daten übertragen und geholt.

STDWNDS.DLG enthält zwei Ressourcendefinitionen: eine Definition für einen Suchen-Dialog und eine Definition für einen Ersetzen-Dialog.

Um ein *TSearchDialog* verwenden zu können, müssen Sie diese Dialogressourcen compilieren und Ihrer ObjectWindows-Anwendung hinzufügen.

Querverweis: *TEditWindow::CMEditFind, TEditWindow::CMEditReplace*

Element-funktion

Konstruktor
```
TSearchDialog(PTWindowsObject AParent, int ResourceId,
              TSearchStruct _FAR &SearchStruct,
              PTModule AModule = NULL);
```

Ruft einen *TDialog*-Konstruktor auf und übergibt einen Zeiger auf das angegebene übergeordnete Fenster (*AParent*).

Erstellt die Steuerelementobjekte *TEdit* und *TCheckBox*, die mit den entsprechenden Suchen-/Ersetzen-Steuerelementobjekten verknüpft werden sollen. Setzt *TransferBuffer* auf die angegebene *TSearchStruct*-Struktur. Der Text und die Optionen werden zwischen dieser Struktur und dem Dialog übertragen.

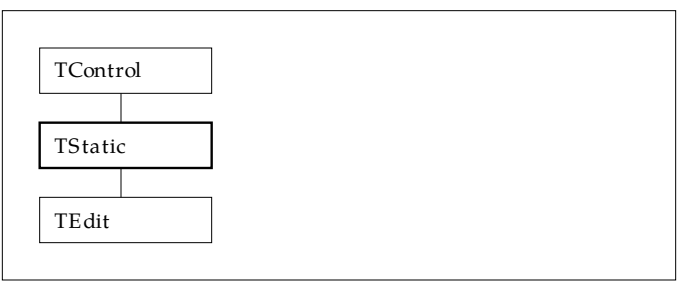
Der Parameter *ResourceId* sollte entweder *SD_SEARCH* oder *SD_REPLACE* sein, entsprechend einer Dialog-Vorlage, die in STDWNDS.DLG definiert ist.

Wenn *ResourceId* gleich *SD_SEARCH* ist, wird ein Suchen-*TSearchDialog* ausgeführt. Ist *ResourceId* gleich *SD_REPLACE*, wird ein Ersetzen-*TSearchDialog* ausgeführt.

Querverweis: *TWindowsObject::TransferBuffer*

TStatic

static.h



TStatic ist ein Schnittstellenobjekt, das statischen Text in Windows repräsentiert. Zur Erstellung eines statischen Steuerelements in einem übergeordneten *TWindow* muß ein *TStatic*-Objekt verwendet werden. *TStatic* vereinfacht auch Textänderungen in statischen Steuerelementen eines *TDialog*-Objekts.

Datenelement

TextLen

```
WORD TextLen;
```

TextLen speichert die Größe des Textpuffers für statische Steuerelemente. Die tatsächliche Anzahl der Zeichen, die in dem statischen Steuerelement gespeichert werden können, ist um 1 niedriger als *TextLen*, weil am Ende des Strings eine abschliessende Null steht. *TextLen* gibt auch die Anzahl der von der Elementfunktion *Transfer* übertragenen Bytes an.

Elementfunktionen

Konstruktor

```
TStatic(PTWindowsObject AParent, int AnId, LPSTR ATitle,
        int X, int Y, int W, int H, WORD ATextLen,
        PTModule AModule = NULL);
```

Erstellt ein statisches Steuerelement mit dem angegebenen übergeordneten Fenster (*AParent*), der Steuerelement-ID (*AnId*), mit dem Text (*ATitle*), mit der Position (*X*, *Y*) relativ zum Ursprung des Client-Bereichs des übergeordneten Fensters, mit der Breite (*W*), der Höhe (*H*) und der Textlänge (*ATextLen*). Das statische Steuerelement ist per Voreinstellung nach der Erstellung sichtbar und sein Text ist linksbündig ausgerichtet. (*Attr.Style* ist auf *WS_CHILD | WS_VISIBLE | WS_GROUP | SS_LEFT* gesetzt). Die Funktion ruft einen *TControl*-Konstruktor auf.

Querverweis: *TControl::TControl*

Konstruktor
```
TStatic(PTWindowsObject AParent, int ResourceId,
        WORD ATextLen, PTModule Module = NULL);
```

Erstellt ein *TStatic*-Objekt, das mit einem statischen Schnittstellenelement eines *TDialog*-Objekts verknüpft werden soll, ruft den *TControl*-Konstruktor mit den entsprechenden Parametern auf und setzt *TextLen* auf *ATextLen*. Schaltet den Datentransfermechanismus durch den Aufruf von *DisableTransfer* aus. Der Parameter *ResourceId* muß einer Ressource für ein statisches Steuerelement entsprechen, die Sie definieren.

Querverweis: *TControl::TControl*

Konstruktor
```
TStatic(StreamableInit);
```
protected

Stream-Konstruktor von *TStatic*. Ruft den Stream-Konstruktor von *TControl* auf. Er wird aufgerufen, wenn ein *TStatic*-Objekt unter Verwendung von Daten aus einem Stream instantiiert wird.

Querverweis: *TControl::TControl*

build
```
static PTStreamable build();
```

Ruft den Konstruktor *TStatic*(*StreamableInit*) auf und erstellt ein Objekt vom Typ *TStatic*, dessen Daten anschließend aus einem Stream eingelesen werden.

Clear
```
void Clear();
```

Löscht den Text des statischen Steuerelements.

GetClassName
```
virtual LPSTR GetClassName();
```
protected

Liefert "STATIC", den Bezeichner, unter dem die Fensterklasse in Windows registriert wird, zurück.

GetText
```
int GetText(LPSTR ATextString, int MaxChars);
```

Holt den Text des statischen Steuerelements und speichert ihn im *ATextString*-Argument, das die in *MaxChars* angegebene Größe hat, und liefert die Anzahl der kopierten Zeichen zurück.

GetTextLen `int GetTextLen();`

Liefert die Textlänge des statischen Steuerelements zurück.

nameOf `virtual Pchar nameOf() const;`

Für Container-Klassen erforderlich. Überschreibt die abstrakte Funktion in der Klasse *Object*. Diese Funktion liefert "TStatic", den Klassenbezeichnungsstring von *TStatic*, zurück.

read `virtual Pvoid read(Ripstream is);` protected

Ruft *TWindow::read* auf, um das Basisobjekt *TWindow* einzulesen. Danach wird *TextLen* eingelesen.

SetText `void SetText(LPSTR ATextString);`

Setzt den Text des statischen Steuerelements auf den in *ATextString* angegebenen String.

Transfer `virtual WORD Transfer(Pvoid DataPtr, WORD TransferFlag);`

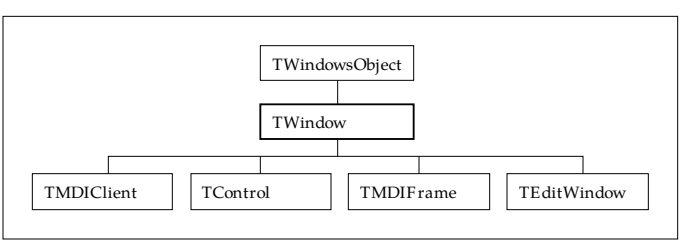
Überträgt soviel Zeichen, wie in *TextLen* angegeben ist, in oder aus dem Transferpuffer, auf den *DataPtr* zeigt. Wenn *TransferFlag* auf *TF_GETDATA* gesetzt ist, wird der Text aus dem statischen Steuerelement in den Puffer übertragen. Wenn *TransferFlag* auf *TF_SETDATA* gesetzt ist, wird der Text des statischen Steuerelements auf den im Transferpuffer stehenden Text gesetzt.

Transfer liefert *TextLen* zurück, also die Anzahl der in einen Puffer geschriebenen oder aus einem Puffer geholten Bytes. Wenn *TransferFlag* auf *TF_SIZEDATA* gesetzt ist, liefert *Transfer* nur *TextLen* zurück, ohne die Daten zu übertragen.

write `virtual void write(Ropstream os);` protected

Ruft *TWindow::write* auf, um das Basisobjekt *TWindow* in den Stream zu schreiben. Danach wird *TextLen* in den Stream geschrieben.

TWindow window.h



TWindow ist von *TWindowsObject* abgeleitet und stellt das Verhalten und die Eigenschaften für Fenster zur Verfügung, wäh-

rend *TDialog* dasselbe nur für Dialoge anbietet. Die *TWindow*-Funktionen sorgen für die Fenstererstellung, einschließlich Registrierung und Zuweisung von Attributen.

TWindow ist ein generisches Fenster (einschließlich Titel), das in seiner Größe verändert und verschoben werden kann. Sie können eine Instanz von *TWindow* initialisieren und erstellen. Normalerweise wird *TWindow* aber als Basis für spezialisierte Fensterklassen eingesetzt.

Sie können eine Instanz von *TWindow* oder einer abgeleiteten Klasse als untergeordnetes MDI-Fenster verwenden, indem Sie einfach ein *TMDIFrame*-Objekt als übergeordnetes Fenster angeben.

Viele der *TWindow*-Elementfunktionen können mit einem *TScroller*-Objekt zusammenarbeiten. Ein *TScroller*-Objekt unterstützt den Bildlauf in einem *TWindow*-Fenster.

TWindow ist außerdem die Basisklasse für *TMDIClient*, einem speziellen Steuerelement für MDI-Anwendungen.

Datenelemente

Attr

```
TWindowAttr Attr;
```

Attr speichert eine *TWindowAttr*-Struktur, welche die Fenster-Erstellungsattribute enthält (die Charakteristika des Schnittstellenelements, das dem Fensterobjekt entspricht). Zu diesen Attributen gehören der Stil, der erweiterte Stil, die Position, die Größe, das Menü-Handle und die Steuerelement-ID des Fensters. In Kapitel 24 »Weitere Komponenten von ObjectWindows« finden Sie die Strukturdefinition von *TWindowAttr*.

Querverweis: *TWindow::TWindow, TWindow::Create*

FocusChild-Handle

```
HANDLE FocusChildHandle
```

FocusChildHandle enthält das Handle des untergeordneten Fensters, das den Fokus hatte, als das Fenster das letzte Mal aktiviert war.

Scroller

```
PTScroller Scroller
```

Wenn *Scroller* ungleich *NULL* ist, zeigt dieses Datenelement auf ein *TScroller*-Objekt, das dem *TWindow*- Objekt Bildlauffähigkeiten verleiht. Das *TScroller*-Objekt wird im Konstruktor von *TWindow* instantiiert. Hier wird auch *Scroller* gesetzt.

Elementfunktionen

Konstruktor

```
TWindow(PTWindowsObject AParent, LPSTR ATitle,
        PTModule AModule = NULL);
```

Erstellt ein Fensterobjekt mit dem in *AParent* angegebenen übergeordneten Fenster. (*AParent* muß für Hauptfenster, die keine übergeordneten Fenster haben, *NULL* sein). Der zugehö-

rige Text – normalerweise der Titel des Fensters – wird aus *ATitle* kopiert. Diese Funktion ruft den *TWindowsObject*-Konstruktor auf und setzt die Positions- und Größen-Felder in der *Attr*-Struktur auf die Voreinstellungen, die für überlappende und Pop-up-Fenster angebracht sind. Wenn *AParent* gleich *NULL* ist, wird *Attr.Style* auf *WS_OVERLAPPEDWINDOW* gesetzt. Für diejenigen *TWindow*-Objekte, die untergeordnete Fenster eines *TMDIFrame*-Objekts sind, wird es auf *WS_CLIPSIBLINGS* gesetzt. Für alle anderen *TWindow*-Objekte wird *Attr.Style* auf *WS_VISIBLE* gesetzt.

Die voreingestellten Werte in der *Attr*-Struktur können im Konstruktor einer von *TWindow* abgeleiteten Klasse (oder jederzeit vor der Erstellung ihres Schnittstellenelements) neu gesetzt werden. Der Konstruktor einer von *TWindow* abgeleiteten Klasse kann außerdem *Scroller* (das per Voreinstellung *NULL* ist) auf ein instantiiertes *TScroller*-Objekt setzen. Sie müssen sicherstellen, daß die Konstruktoren von abgeleiteten Klassen den *TWindow*-Konstruktor aufrufen.

Wenn *AParent* ein MDI-Rahmenfenster ist, wird dieses als untergeordnetes MDI-Fenster erstellt (das Flag *WB_MDICHILD* ist gesetzt). Wenn das neue *TWindow*- Objekt ein Pop-up-Fenster ist (und deshalb nicht als untergeordnetes MDI-Fenster erstellt werden sollte), muß *SetFlags*(*WB_MDICHILD,FALSE*) nach der Initialisierung von this und vor der Erstellung des Fensters aufgerufen werden.

Querverweis: *TWindowsObject::TWindowsObject*

Konstruktor

```
TWindow(HWND AnHWindow, PTModule AModule = NULL);
```

Konstruktor für ein *TWindow*, das in einer DLL ein Nicht-ObjectWindows-Fenster repräsentiert. Fragt zur Initialisierung der Datenelemente Windows ab.

Konstruktor

```
TWindow(StreamableInit);                                    protected
```

Stream-Konstruktor von *TWindow*. Ruft den Stream-Konstruktor von *TWindowsObject* auf. Er wird aufgerufen, wenn ein *TWindow*-Objekt unter Verwendung von Daten aus einem Stream instantiiert wird.

Querverweis: *TWindowsObject::TWindowsObject*

Destruktor

```
virtual ~TWindow();
```

Gibt *Attr.Menu* frei und löscht *Scroller*, wenn er ungleich *NULL* ist.

Activation-Response

```
virtual void ActivationResponse(WORD Activated,
                                BOOL IsIconified);
```

Wird von *WMActivate* und *WMMDIActivate* aufgerufen, um für die Steuerelemente eines Fensters eine Tastaturschnittstelle zur

Verfügung zu stellen. Schaltet durch den Aufruf von *TWindowsObject::ActivationResponse* die Tastaturbehandlung aus oder ein. Wenn das Fenster deaktiviert wird (wenn *Activated* gleich Null ist), speichert diese Funktion das Handle des untergeordneten Steuerelements, das aktuell den Fokus hat, in *FocusChildHandle*. Wenn das Fenster aktiviert wird, wird der Fokus für das untergeordnete Fenster gesetzt, dessen Handle gespeichert wurde.

Querverweis: *TWindowsObject::ActivationResponse, TWindows-Object::EnableKBHandler, TWindow::WMMDIActivate*

AssignMenu `virtual BOOL AssignMenu(LPSTR MenuName);`

Setzt *Attr.Menu* auf den angegebenen Namen in *MenuName*. Gibt gegebenenfalls den String frei, auf den *Attr.Menu* zuvor zeigt. Wenn *HWindow* ungleich Null ist, wird das Menü des Fensters geladen und gesetzt, wobei ein eventuell vorher vorhandenes Menü gelöscht wird.

AssignMenu `virtual BOOL AssignMenu(int MenuId);`

Ermittelt die Menü-ID des Fensters mit Hilfe des Integer-Ressourcen-Bezeichners, der in einer Menüressourcendatei definiert ist. Die Funktion übergibt den Wert an die andere *AssignMenu*-Elementfunktion, die dann die eigentliche Arbeit macht.

build `static PTStreamable build();`

Ruft den Konstruktor *TWindow(StreamableInit)* auf und erstellt ein Objekt vom Typ *TWindow*, dessen Datenelemente anschließend aus einem Stream eingelesen werden.

Querverweis: *TWindow::TWindow*

Create `virtual BOOL Create();`

Erstellt ein Schnittstellenelement und verknüpft es mit dem *TWindow*-Objekt. Die Verknüpfung wird nicht hergestellt, wenn *Status* ungleich Null ist. *Create* ruft zuerst *Register* auf, um die Fensterklasse zu registrieren (wenn sie nicht bereits registriert ist). Danach wird das Schnittstellenelement erstellt und verknüpft.

Wenn das Flag *WB_FROMRESOURCE* gesetzt ist, wird nur die Verknüpfung hergestellt, Registrierung und Erstellung unterbleiben, weil das Schnittstellenelement bereits durch eine Ressourcendefinition erstellt wurde. Danach wird *SetupWindow* aufgerufen, wodurch das neu erstellte Fenster aufgebaut wird.

Nach erfolgreicher Durchführung liefert *Create TRUE* zurück. Andernfalls wird *Error* aufgerufen und *FALSE* zurückgeliefert.

Normalerweise werden Sie *Create* nicht direkt aufrufen. *Create* wird von *TModule::MakeWindow* aufgerufen, das zuerst Gültigkeitsprüfungen durchführt.

Querverweis: *TWindowsObject::Register, TWindowsObject::SetupWindow, TModule::MakeWindow*

GetClassName `virtual LPSTR GetClassName();` protected

Liefert den in Windows registrierten Bezeichner der Fensterklasse, "OWLWindow", für ein *TWindow*-Objekt zurück.

GetWindowClass `virtual void GetWindowClass(WNDCLASS _FAR & AWndClass);` protected

Füllt die *WNDCLASS*-Struktur, die in *AWndClass* angegeben ist, mit den standardmäßigen Registrierungsattributen für ein *TWindow*-Objekt aus. Das Datenelement *Style* wird auf *CS_HREDRAW* oder *CS_VREDRAW* gesetzt. Das Symbol wird auf ein generisches Symbol gesetzt, der Mauszeiger auf den vordefinierten Mauspfeilzeiger und die Hintergrundfarbe auf die Hintergrundfarbe des Systemfensters. Der Name der Klasse, die registriert werden soll, wird durch einen Aufruf von *GetClassName* ermittelt.

Querverweis: *TWindow::GetClassName, TWindowsObject::Register, TWindow::Create*

isA `virtual classType isA() const;`

Für Container-Klassen erforderlich. Überschreibt die abstrakte Funktion in der Klasse *Object*. Diese Funktion liefert *WindowClass*, den Klassenbezeichner von *TWindow*, zurück.

nameOf `virtual Pchar nameOf() const;`

Für Container-Klassen erforderlich. Überschreibt die abstrakte Funktion in der Klasse *Object*. Diese Funktion liefert "TWindow", den Klassenbezeichnungsstring von *TWindow*, zurück.

Paint `virtual void Paint(HDC PaintDC, PAINTSTRUCT _FAR & PaintInfo);` protected

Dient als Platzhalter für abgeleitete Klassen, die diese Funktion *Paint* überschreiben. *Paint* wird von *WMPaint* aufgerufen, wenn Windows anfordert, daß der Inhalt des Fensters neu gezeichnet werden soll. Die Funktion verwendet *PaintDC* als Bildschirmkontext für Text- und Grafik-Ausgabefunktionen. Die *PaintInfo*-Struktur enthält Informationen über die Zeichnenoperation einschließlich des Bereichs, der gezeichnet werden muß.

Querverweis: *TWindow::WMPaint*

read `virtual Pvoid read(Ripstream is);` protected

Ruft *TWindowsObject::read* auf, um das Basisobjekt *TWindows-*

Object einzulesen. Wenn das Flag *WB_FROMRESOURCE* nicht gesetzt ist, werden danach *Attr.Style*, *Attr.ExStyle*, *Attr.X*, *Attr.Y*, *Attr.W*, *Attr.H* und *Attr.Param* und danach *Attr.Id*, *Attr.Menu* und *Scroller* eingelesen. *FocusChildHandle* wird auf 0 gesetzt.

SetupWindow `virtual void SetupWindow();` protected

Baut das neu erstellte Fenster auf und ruft *TWindowsObject::SetupWindow* zur Erstellung der zugehörigen untergeordneten Fenster auf. Wenn der Anwender die Eingaben über die Tastatur vornimmt und *FocusChildHandle* nicht gesetzt wurde, wird es auf das geeignete untergeordnete Fenster gesetzt. Wenn das Fenster ein untergeordnetes MDI-Fenster ist, ruft *SetupWindow* die Funktion *SetFocus* auf, um dem neuen Fenster den Fokus zu übergeben. Wenn das Fenster ein *TScroller*-Objekt hat, ruft *SetupWindow* die Funktion *SetSBarRange* auf, um den Bereich der Bildlaufleisten festzulegen.

Die Funktion kann in abgeleiteten Klassen überschrieben werden, so daß sie zusätzliche Initialisierungsaufgaben ausführt. Eine Neudefinition in einer abgeleiteten Klasse sollte die *SetupWindow*-Funktion der Basisklasse aufrufen.

Querverweis: *TScroller::SetSBarRange*

WMActivate `virtual void WMActivate(RTMessage Msg) = [WM_FIRST + WM_ACTIVATE];` protected

Stellt eine Tastaturschnittstelle für die Steuerelemente eines Fensters zur Verfügung. Wenn das Fenster deaktiviert wird, speichert diese Funktion das Handle des untergeordneten Steuerelements, das aktuell den Fokus hat, in *FocusChildHandle*. Wenn das Fenster aktiviert wird, wird der Fokus auf das untergeordnete Fenster gesetzt, dessen Handle gespeichert ist.

Querverweis: *TWindowsObject::EnableKBHandler*

WMCreate `virtual void WMCreate(RTMessage Msg) = [WM_FIRST + WM_CREATE];` protected

Antwortet mit dem Aufruf von *SetupWindow* auf die *WM_CREATE*-Botschaft.

Querverweis: *TWindow::SetupWindow, TWindow::Create*

WMHScroll `virtual void WMHScroll(RTMessage Msg) = [WM_FIRST + WM_HSCROLL];` protected

Antwortet auf horizontale Bildlaufereignisse. Wenn die Botschaft von einem Bildlaufleisten-Steuerelement stammt, wird *DispatchScroll* aufgerufen. Andernfalls – wenn *Scroller* ungleich Null ist – wird *Scroller->HScroll* aufgerufen. (Dabei wird auf Grund eines Windows-Fehlers unterstellt, daß in einem Fenster mit Bildlaufleisten-Stil keine Bildlaufleisten-Steuerelemente vorhanden sind.)

Querverweis: *TScroller::HScroll*

WMLButtonDown

```
virtual void WMLButtonDown(RTMessage Msg) =
[WM_FIRST + WM_BUTTONDOWN];                    protected
```

Botschaftsantwortfunktion für eine eingehende *WM_LBUTTONDOWN*-Botschaft. Wenn *TWindow* einen automatischen Scroller hat, wird die normale Botschaftsbearbeitung weitergeführt, bis eine *WM_LBUTTONUP*-Botschaft eingeht, und *Scroller->AutoScroll* aufgerufen. Fängt vorübergehend alle Maueingaben ab.

Wenn Sie diese Elementfunktion überschreiben wollen, um Mauseingaben zu bearbeiten, Sie aber weiterhin den automatischen Bildlauf beibehalten wollen, müssen Sie diese Funktion von der neu definierten Funktion *WMLButtonDown* aus aufrufen.

WMMDIActivate

```
virtual void WMMDIActivate(RTMessage Msg) =
[WM_FIRST + WM_MDIACTIVATE];                   protected
```

Antwortet auf eine eingehende *WM_MDIACTIVATE*-Botschaft. Die Funktion ruft zunächst *DefWndProc* auf, setzt dann *ActiveChild* im übergeordneten Fenster und ruft schließlich *ActivationResponse* auf, um für die Steuerelemente eines Fensters eine Tastaturschnittstelle zur Verfügung zu stellen.

Wenn Sie *WMMDIActivate* in einer abgeleiteten Klasse überschreiben, muß *WMMDIActivate* der Basisklasse unmittelbar nach der anderen Bearbeitung aufgerufen werden (außer es besteht ein spezieller Grund für einen Aufruf an einer anderen Stelle).

Querverweis: *TWindow::ActivationResponse*

WMMove

```
virtual void WMMove(RTMessage Msg) =
[WM_FIRST + WM_MOVE];                          protected
```

Antwortet auf eine *WM_MOVE*-Botschaft. Dabei werden die Koordinaten der neuen Fensterposition – wenn das Fenster nicht als Symbol dargestellt oder maximal vergrößert ist – in den Elementen *X* und *Y* der *Attr*-Struktur gespeichert.

WMPaint

```
virtual void WMPaint(RTMessage Msg) =
[WM_FIRST + WM_PAINT];                         protected
```

Antwortet auf eine *WM_PAINT*-Botschaft. Wenn das Fenster eine vordefinierte Windows-Fensterklasse hat, wird *DefWndProc* aufgerufen. Andernfalls erfolgt ein Aufruf von *Paint*, damit vor und nach dem Aufruf von *Paint* das für Windows erforderliche Aufbauen und Löschen (*BeginPaint* und *EndPaint*) durchgeführt wird. Wenn zum Fenster ein *TScroller*-Objekt gehört hat, werden vor und nach dem Aufruf von *Paint* dessen Elementfunktionen *BeginView* und *EndView* aufgerufen.

Querverweis: *TMDIClient::WMPaint*, *TWindow::Paint*, *TScroller::BeginView*, *TScroller::EndView*

WMSize `virtual void WMSize(RTMessage Msg) = [WM_FIRST + WM_SIZE];` protected

Antwortet bei Fenstern mit Bildlaufleisten auf eine Größenänderung des Fensters, indem die Funktion *SetPageSize* aufgerufen wird. Bei Fenstern mit oder ohne Scroller wird die Größe des Fensters in *Attr* gespeichert.

Querverweis: *TScroller::SetPageSize*

WMVScroll `virtual void WMVScroll(RTMessage Msg) = [WM_FIRST + WM_VSCROLL];` protected

Wenn die Botschaft von einem Bildlaufleisten-Steuerelement stammt, wird *DispatchScroll* aufgerufen. Andernfalls – wenn *Scroller* ungleich Null ist – wird *Scroller->VScroll* aufgerufen. (Dabei wird auf Grund eines Windows-Fehlers unterstellt, daß in einem Fenster mit Bildlaufleisten-Stil keine Bildlaufleisten-Steuerelemente vorhanden sind.)

Querverweis: *TScroller::VScroll*

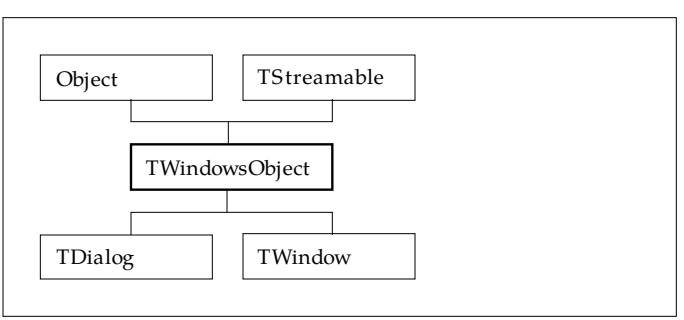
write `virtual void write(Ropstream os);` protected

Ruft *TWindowsObject::write* auf, um zuerst das Basisobjekt *TWindowsObject* in den Stream zu schreiben.

Wenn das Flag *WB_FROMRESOURCE* nicht gesetzt ist, werden danach *Attr.Style*, *Attr.ExStyle*, *Attr.X*, *Attr.Y*, *Attr.W*, *Attr.H* und *Attr.Param* und danach *Attr.Id*, *Attr.Menu* und *Scroller* in den Stream geschrieben.

TWindowsObject

owindobj.h



TWindowsObject ist eine von *Object* und *TStreamable* abgeleitete abstrakte Klasse und definiert das grundsätzliche Verhalten aller Schnittstellenobjekte von ObjectWindows (Fenster, Dialoge und Steuerelemente).

TWindowsObject verwaltet in einer Liste seine untergeordneten Fenster, für die es Verhaltensweisen und Eigenschaften zur Verfügung stellt. Beispielsweise werden bei der Erstellung

einer *TWindowsObject*-Instanz gleichzeitig die zugehörigen untergeordneten Fenster erzeugt.

Das Schließen von *TWindowsObject* ist nur dann möglich, wenn die zugehörigen untergeordneten Fenster ebenfalls geschlossen werden können. *TWindowsObject* definiert außerdem Elementfunktionen, die den Zugriff auf die untergeordneten Fenster in der zugehörigen Liste ermöglichen.

TWindowsObject spielt außerdem eine wichtige Rolle bei der Verteilung eingehender Botschaften zu Botschaftsantwortfunktionen.

Datenelemente

DefaultProc `FARPROC DefaultProc;` protected

Enthält die Adresse der Standard-Fensterprozedur. Diese definiert die Standardbearbeitung der Windows-Botschaften.

HWindow `HWND HWindow;`

HWindow enthält ein Handle für das mit dem Schnittstellenobjekt verknüpfte Schnittstellenelement. Falls kein solches Element existiert, enthält *HWindow* 0. *HWindow* wird bei der Erstellung des zugehörigen Schnittstellenelements auf dessen Handle und bei der Freigabe des Elements auf 0 gesetzt.

Parent `PWindowsObject Parent;`

Zeigt auf das übergeordnete Fenster des Schnittstellenobjekts.

Status `int Status;`

Status meldet einen Fehler, der bei der Initialisierung eines Schnittstellenobjekts auftritt.

Ist *Status* ungleich 0, war die Initialisierung des Objekts nicht erfolgreich. Von *TWindowsObject* abgeleitete Klassen versuchen nicht, ein Schnittstellenelement mit einem Objekt zu verknüpfen, dessen Initialisierung erfolglos war. Diese Klassen setzen *Status* auch dann, wenn der Versuch der Zuordnung eines Schnittstellenelements fehlschlägt.

ObjectWindows definiert Fehlerwerte wie *EM_INVALIDWINDOW*, *EM_INVALIDCLIENT*, *EM_INVALIDCHILD* und *EM_INVALIDMAINWINDOW* (vergleichen Sie dazu die »EM_XXXX-Konstanten« in Kapitel 24 »Weitere Komponenten von ObjectWindows«). Sie können *Status* auch auf Fehlerwerte setzen, die Sie eigens zur Kennzeichnung von Initialisierungsfehlern definieren. *Status* wird von *Create* oder *SetupWindow* auf Fehlerwerte gesetzt.

Title `LPSTR Title;`

Zeigt auf den Titel des Fensters.

Querverweis: *TDialog::SetCaption*, *TDialog::SetupWindow*, *TWindowsObject::SetCaption*

TransferBuffer `Pvoid TransferBuffer;` protected

TransferBuffer zeigt auf einen Puffer zur Übertragung von Daten in und aus dem *TWindowsObject*.

Der Puffer enthält normalerweise Daten, die von den Fenstern der zu *TWindowsObject* gehörigen Liste untergeordneter Fenster benutzt werden. Ist *TransferBuffer NULL*, sind keine Daten zu übertragen.

Element-funktionen

Konstruktor
```
TWindowsObject(PTWindowsObject AParent,
               PTModule AModule = NULL);
```

Setzt *Parent* auf *AParent*. Ist *AParent* ungleich Null, wird this zur Liste der untergeordneten Fenster von *AParent* hinzugefügt und *EnableAutoCreate* aufgerufen. this wird dann zusammen mit *AParent* erstellt und angezeigt. Wenn Ableitungen von *TWindowsObject* an dynamische Linkbibliotheken übergeben werden (explizit oder implizit), erfolgt dies am besten explizit in *AModule*.

AModule bestimmt das Anwendungs- oder DLL-Modul, das die *TWindowsObject*-Instanz besitzt. ObjectWindows benötigt den korrekten Wert von *AModule*, um die erforderlichen Ressourcen finden zu können. Wenn *AModule* Null ist (voreingestellter Wert), setzt *TWindowsObject* sein Modul (das mit *GetModule* ermittelt werden kann) nach folgenden Regeln:

- Wenn der *TWindowsObject*-Konstruktor von einer Anwendung aus aufgerufen wird, wird das Modul auf die Anwendung gesetzt.

- Wenn der *TWindowsObject*-Konstruktor von einer dynamischen Linkbibliothek aus aufgerufen wird, die dynamisch mit der ObjectWindows-DLL gelinkt wird, und die aktuell ausführende Anwendung auf dieselbe Art und Weise gelinkt wird, wird das Modul auf die aktuell ausgeführte Anwendung gesetzt. (In Kapitel 17 »Dynamische Linkbibliotheken« dieses Handbuchs finden Sie Programmieranweisungen für den Einsatz von ObjectWindows als dynamische Linkbibliothek.)

- Wenn der *TWindowsObject*-Konstruktor von einer dynamischen Linkbibliothek aus aufgerufen wird, die statisch mit der ObjectWindows-Bibliothek gelinkt wird (z.B. OWLWL.LIB) oder wenn die aufrufende dynamische Linkbibliothek zwar dynamisch mit der ObjectWindows-DLL gelinkt ist, die aktuell ausführende Anwendung jedoch nicht, wird für das Setzen des Moduls keine Voreinstellung verwendet. Stattdessen wird *Status* auf *EM_INVALIDMODULE* gesetzt und die Erstellung des Objekts schlägt fehl.

Querverweis: *TWindowsObject::EnableAutoCreate*

Konstruktor `TWindowsObject(StreamableInit);` protected

Stream-Konstruktor von *TWindowsObject*. Er wird aufgerufen, wenn eine abgeleitete Klasse unter Verwendung von Daten aus einem Stream instantiiert wird.

Destruktor `virtual ~TWindowsObject();`

Ruft *Destroy* auf, um ein noch zugeordnetes Schnittstellenelement freizugeben und zu entfernen. Löscht die Fensterobjekte in der zugehörigen Liste der untergeordneten Fenster und entfernt anschließend this aus der entsprechenden Liste von *Parent*.

build `static PTStreamable build();`

Ruft den Konstruktor *TWindowsObject*(*StreamableInit*) auf und erstellt ein Objekt vom Typ *TWindowsObject*, dessen Datenelemente anschließend aus einem Stream eingelesen werden.

ActivationResponse
```
virtual void ActivationResponse(WORD Activated,
                                BOOL IsIconified);
```

Wird von *TWindowsObject::WMActivate* und *TWindow::ActivationResponse* aufgerufen. Wenn this aktiviert wurde (wenn *Activated* ungleich Null ist), und die Tastaturbehandlung für this angefordert wird, wird diese durch den Aufruf von *GetApplication*()->*SetKBHandler* ein-, andernfalls ausgeschaltet.

IsIconified ist *TRUE*, wenn das Fenster als Symbol dargestellt ist, andernfalls *FALSE*.

Querverweis: *TWindowsObject::WMActivate, TWindowsObject::EnableKBHandler, TApplication::SetKBHandler*

AfterDispatchHandler
```
virtual void AfterDispatchHandler();
```

Überschreiben Sie diese Funktion zur Nachbearbeitung einer eingegangenen Botschaft (diese Funktion führt per Voreinstellung nichts aus). Sie wird nach einem Aufruf einer Botschaftsantwort von *DispatchAMessage* aufgerufen.

Querverweis: *TWindowsObject::DispatchAMessage*

BeforeDispatchHandler
```
virtual void BeforeDispatchHandler();
```

Überschreiben Sie diese Funktion zur Vorbearbeitung einer eingegangenen Botschaft (diese Funktion führt per Voreinstellung nichts aus). Sie wird vor einem Aufruf einer Botschaftsantwort von *DispatchAMessage* aufgerufen.

Querverweis: *TWindowsObject::DispatchAMessage*

CanClose `virtual BOOL CanClose();`

Liefert *TRUE* zurück, wenn das zugehörige Schnittstellenelement geschlossen werden kann. Ruft die *CanClose*-Element-

funktion jedes untergeordneten Fensters auf. Liefert *FALSE* zurück, wenn *FALSE* von einem der *CanClose*-Aufrufe zurückgeliefert wurde.

ChildWithID

```
PTWindowsObject ChildWithId(int Id);
```

Liefert einen Zeiger auf das Fenster in der Liste mit untergeordneten Fenstern zurück, dessen ID angegeben wurde. Ist kein untergeordnetes Fenster mit der angegebenen *Id* in der Liste enthalten, wird *NULL* zurückgeliefert.

CloseWindow

```
void CloseWindow();
```

Ruft *ShutDownWindow* auf, nachdem festgestellt wurde, daß dies möglich ist. Ist this das Hauptfenster der Anwendung, wird *GetApplication*()->*CanClose* aufgerufen. Andernfalls wird durch einen Aufruf von *this->CanClose* ermittelt, ob das Fenster geschlossen werden kann.

Querverweis: *TWindowsObject::CanClose*, *TApplication::CanClose*, *ShutDownWindow, WMClose*

CMExit

```
virtual void CMExit (RTMessage Msg) =
[CM_FIRST + CM_EXIT];                                    protected
```

Wird als Antwort auf die Auswahl eines Menüeintrags mit der ID *CM_EXIT* aufgerufen. Ist this das Hauptfenster, erfolgt ein Aufruf von *CloseWindow*.

Create

```
virtual BOOL Create() = 0;
```

Abstrakte Funktion, die in abgeleiteten Klassen überschrieben wird, um das mit dem ObjectWindows-Schnittstellenobjekt verknüpfte Windows-Schnittstellenelement zu erstellen.

CreateChildren

```
BOOL CreateChildren();
```

Erstellt die untergeordneten Fenster aus der entsprechenden Liste, deren Flag (*WB_AUTOCREATE*) zur automatischen Erstellung gesetzt ist.

Querverweis: *TWindowsObject::EnableAutoCreate*, *TWindowsObject::DisableAutoCreate*

DefChildProc

```
virtual void DefChildProc(RTMessage Msg);                protected
```

Führt die Standardbearbeitung für eine Botschaft aus, die auf der ID untergeordneter Fenster basiert. *DefChildProc* ruft standardmäßig *DefWndProc* auf.

Sie können diese Elementfunktion überschreiben, damit sie Botschaften von allen untergeordneten Fenstern beantwortet, für die keine entsprechende, auf der ID des untergeordneten Fensters basierende Antwort definiert wurde. Eine derartige Bearbeitung von Botschaften untergeordneter Fenster ist in

manchen Fällen sinnvoller, als für jedes untergeordnete Fenster eine besondere Botschaftsantwortfunktion zu definieren.

DefCommandProc

```
virtual void DefCommandProc(RTMessage Msg);
```

protected

Führt die Standardbearbeitung für eine eingehende Befehlsbotschaft aus. *DefCommandProc* überprüft den Adressaten der Botschaft. Ist dies das Fenster selbst, wird *DefWndProc* aufgerufen. Andernfalls wird die Botschaft an das übergeordnete Fenster übergeben, oder, wenn es das nicht gibt, an den ursprünglichen Adressaten.

Sie können diese Elementfunktion überschreiben, damit sie auf Befehlsbotschaften antwortet, für die keine entsprechende auf Befehlen basierende Botschaftsantwortfunktion definiert wurde. Eine derartige Bearbeitung von Befehlsbotschaften ist in manchen Fällen sinnvoller, als für jeden Befehl eine Botschaftsantwortfunktion zu definieren.

DefNotificationProc

```
virtual void DefNotificationProc(RTMessage Msg);
```

protected

Führt die Standardbearbeitung für eine von this generierte Benachrichtigungsbotschaft aus. Die Botschaft wird dazu als eine auf der ID eines untergeordneten Fensters basierende Botschaft an das übergeordnete Fenster übergeben (this kann seine eigenen Benachrichtungsbotschaften beantworten).

Sie können diese Elementfunktion überschreiben, damit sie alle von this gesendeten Benachrichtigungsbotschaften beantwortet. Eine derartige Bearbeitung von Benachrichtigungsbotschaften ist in manchen Fällen sinnvoller, als für jede entsprechende Botschaft eine auf Benachrichtigung basierende Botschaftsantwortfunktion zu definieren.

DefWndProc

```
virtual void DefWndProc(RTMessage Msg);
```

Legt die Standardbearbeitung für eine eingehende Botschaft fest und benutzt dazu die von Windows definierte Standard-Botschaftsbearbeitung.

DefWndProc ruft die voreingestellte Fensterprozedur eines Fensters auf und setzt *Msg.Result* auf den zurückgelieferten Wert. Ist this ein Dialogfenster, wird *Msg.Result* auf 0 gesetzt, damit Windows die Standardbearbeitung anwenden kann.

Destroy

```
virtual void Destroy;
```

Gibt das zugehöriges Schnittstellenelement frei, nachdem *EnableAutoCreate* für jedes Fenster in der Liste der untergeordneten Fenster aufgerufen wurde. Dadurch wird sichergestellt, daß bei einer Neuerstellung von this die Fenster dieser Liste ebenfalls neu erstellt werden.

Querverweis: *TWindowsObject::WMDestroy* , *TWindowsObject::EnableAutoCreate*

DisableAutoCreate

```
void DisableAutoCreate();
```

Bewirkt, daß untergeordnete Fenster nicht gleichzeitig mit dem übergeordneten Fenster erzeugt und angezeigt werden. Rufen Sie *DisableAutoCreate* für Pop-up-Fenster und Steuerelemente auf, wenn Sie diese nach ihrem übergeordneten Fenster erstellen und anzeigen wollen.

Querverweis: *TWindowsObject::EnableAutoCreate*

DisableTransfer

```
void DisableTransfer();
```

Deaktivert den Transfermechanismus für das Schnittstellenobjekt, der die Übertragung von Statusdaten zum und vom Transferpuffer ermöglicht.

DispatchAMessage

```
virtual void DispatchAMessage(WORD AMsg, RTMessage
                              AMessage, void
                              (TWindowsObject::* _FAR)
                              (RTMessage));
```

DispatchAMessage gibt eine eingehende Windows-Botschaft an die entsprechende Botschaftsantwortfunktion weiter. Ist keine Antwort definiert, wird stattdessen die angegebene Standard-Elementfunktion aufgerufen. Die Funktion ist beim Setzen von Breakpoints während des Debuggens hilfreich.

Sie ruft vor und nach der Verteilung von Botschaften *BeforeDispatchHandler* und *AfterDispatchHandler* auf.

DispatchScroll

```
void DispatchScroll(RTMessage Msg);
```

protected

Wird von *WMHScroll* und *WMVScroll* zur Weitergabe von Bildlaufleisten-Botschaften aufgerufen.

Querverweis: *TWindowsObject::WMHScroll* , *TWindowsObject::WMVScroll*

DrawItem

```
virtual void DrawItem(DRAWITEMSTRUCT _FAR & DrawInfo);
```

Überschreiben Sie diese Funktion, um festzulegen, wie ein benutzergezeichnetes Steuerelement oder Menüeintrag gezeichnet wird. *DrawItem* wird aufgerufen, wenn ein Steuerelement, das sich nicht selbst zeichnen kann, oder ein Menüeintrag gezeichnet werden muß. Diese Elementfunktion wird auch aufgerufen, wenn sich der Auswahl- oder der Fokus-Status des Steuerelements bzw. des Eintrags ändert, so daß seine Darstellung entsprechend geändert werden kann. Wenn es die entsprechenden Funktionen überschreibt, kann sich ein benutzergezeichnetes Steuerelement auch selbst zeichnen.

EnableAutoCreate

```
void EnableAutoCreate();
```

Stellt sicher, daß ein untergeordnetes Fenster-Schnittstellenelement zusammen mit seinem übergeordneten Fenster erstellt und angezeigt wird. *EnableAutoCreate* ist für Fenster und Steue-

relemente standardmäßig aktiviert, für Dialogfenster dagegen ausgeschaltet. Rufen Sie *EnableAutoCreate* auf, wenn ein Dialogfenster zusammen mit seinem übergeordneten Fenster erstellt und angezeigt werden soll.

Querverweis: *TWindowsObject::DisableAutoCreate*

EnableKB-Handler

```
void EnableKBHandler();
```

Aktiviert eine Tastaturschnittstelle für die Steuerelemente eines Fensters oder eines nicht-modalen Dialogs, damit der Anwender Steuerelemente mit Tab - und Pfeiltasten anwählen kann. Die Tastaturschnittstelle ist für Fenster und Dialogfenster standardmäßig ausgeschaltet. (Diese Schnittstelle wird von Windows für Dialogfenster zur Verfügung gestellt.)

Querverweis: *TWindowsObject::WMActivate , TApplication::-GetKBHandler*

EnableTransfer

```
void EnableTransfer();
```

Aktiviert den Transfermechanismus, der die Übertragung von Statusdaten zwischen dem Fenster und einem Transferpuffer ermöglicht.

FirstThat

```
PTWindowsObject FirstThat(TCondFunc Test,
                          Pvoid PParamList);
```

Iteriert über die Liste der untergeordneten Fenster und ruft eine Boolesche *Test*-Funktion auf, wobei letzterer Funktion jedes untergeordnete Fenster und *PParamList* als Argumente übergeben werden. Wenn ein *Test*-Aufruf *TRUE* zurückliefert, wird die Iteration beendet und *FirstThat* gibt das untergeordnete Fensterobjekt zurück, das an *Test* übergeben wurde. Andernfalls liefert *FirstThat NULL* zurück. (Informationen zu *TCondFunc* finden Sie in Kapitel 24 »Weitere Komponenten von ObjectWindows«.)

Im folgenden Beispiel ruft *GetFirstChecked FirstThat* auf, um einen Zeiger (*P*) auf das erste markierte Auswahlfeld in der Liste der untergeordneten Fenster zu erhalten:

```
BOOL IsThisBoxChecked(Pvoid P, Pvoid) {
  return ((PTCheckBox)P->GetCheck() == BF_CHECKED);
}
PTCheckBox TMyWindow::GetFirstChecked(); {
  return (FirstThat(IsThisBoxChecked, NULL);
}
```

FirstThat

```
PTWindowsObject FirstThat(TCondMemFunc Test,
                          Pvoid PParamList);
```

Vergleichen Sie hierzu die Beschreibung der obigen *FirstThat*-Funktion. Der Unterschied zwischen den beiden *FirstThat*-Funktionen beruht darin, daß die oben beschriebene Funktion

eine Funktion und die hier angesprochene Funktion eine Elementfunktion als Parameter hat. Die *Test*-Funktion hat also Zugriff auf die Elemente des Objekts. In Kapitel 24 »Weitere Komponenten von ObjectWindows« finden Sie eine Erläuterung von *TCondMemFunc*.

ForEach `void ForEach(TActionFunc Action, Pvoid PParamList)`

Iteriert über die Liste der untergeordneten Fenster und ruft die angegebene Funktion (*Action*) auf, wobei jedes untergeordnete Fenster als Argument in *PParamList* übergeben wird. (Informationen zu *TActionFunc* finden Sie in Kapitel 24 »Weitere Komponenten von ObjectWindows«.) Im folgenden Beispiel ruft *CheckAllBoxes ForEach* auf, um alle Auswahlfelder in der Liste der untergeordneten Fenster zu markieren:

```
void CheckTheBox(Pvoid P, Pvoid) {
  (PTCheckBox)P->Check();
}
void CheckAllBoxes(); {
  ForEach(CheckTheBox, NULL);
}
```

ForEach `void ForEach(TActionMemFunc Action, Pvoid PParamList);`

Vergleichen Sie hierzu die Beschreibung der obigen *ForEach*-Funktion. Der Unterschied zwischen den beiden *ForEach*-Funktionen beruht darin, daß die oben beschriebene Funktion eine Funktion und die hier angesprochene Funktion eine Elementfunktion als Parameter hat. *Action* hat also Zugriff auf die Elemente des Objekts. In Kapitel 24 »Weitere Komponenten von ObjectWindows« finden Sie eine Erläuterung von *TActionMemFunc*.

GetApplication `PTApplication GetApplication();`

Erhält einen Zeiger auf das *TApplication*-Objekt, das mit this verknüpft ist.

GetChildPtr `void GetChildPtr(Ripstream is, RPTWindowsObject P);`
protected

Liest aus dem angegebenen Stream eine Referenz auf einen Zeiger auf ein untergeordnetes Fenster ein. Rufen Sie *GetChildPtr* auf, wenn Sie einen Zeiger auf ein untergeordnetes Fenster einlesen wollen, der mit *PutChildPtr* in einen Stream geschrieben wurde.

Querverweis: *TWindowsObject::GetSiblingPtr, TWindowsObject::PutSiblingPtr, TWindowsObject::PutChildPtr*

GetChildren `void GetChildren(Ripstream is);`

Liest untergeordnete Fenster aus dem angegebenen Stream in die entsprechende Liste ein. *GetChildren* sollte nur während

einer *read*-Operation zum Einlesen untergeordneter Fenster aufgerufen werden, die mit *PutChildren* in den Stream geschrieben wurden.

Wichtig: Diese Elementfunktion unterstellt, daß die Liste mit den untergeordneten Fenstern momentan keine Einträge enthält.

GetClassName `virtual LPSTR GetClassName() = 0;` protected

Abstrakte Funktion, die von einer abgeleiteten Klasse überschrieben werden muß. Liefert den Bezeichner, unter dem die Fensterklasse in Windows registriert wird, zurück.

Querverweis: *TWindowsObject::GetWindowClass*

GetClient `virtual PTMDIClient GetClient();`

Liefert *NULL* für alle Nicht-MDI-Schnittstellenobjekte zurück, zu denen es keine MDI-Client-Fenster gibt. *TMDIFrame* überschreibt diese Elementfunktion, damit sie einen Zeiger auf das MDI-Client-Fenster zurückliefert.

GetFirstChild `PTWindowsObject GetFirstChild();` protected

Liefert einen Zeiger auf das erste untergeordnete Fenster in der entsprechenden Liste des Schnittstellenobjekts.

GetId `virtual int GetId();`

Liefert die ID, die für das Auffinden des Fensters in der Liste der untergeordneten Fenster verwendet wird. *GetId* wird von *TControl*-Nachkommen so überschrieben, daß sie den Ressourcenbezeichner (*Attr.Id*) des Objekts zurückliefert.

Per Voreinstellung liefert *TWindowsObject::GetId* 0 zurück. Dadurch werden alle Fenster ohne Steuerelemente, die eine ID von 0 haben, verborgen. Wenn Sie einzelne Fenster adressieren wollen, überschreiben Sie *GetId* so, daß es eine Zahl ungleich Null zurückliefert.

Querverweis: *TControl::GetId*

GetInstance `FARPROC GetInstance();`

Liefert die Instanz des Fensters zurück.

GetLastChild `PTWindowsObject GetLastChild();`

Liefert einen Zeiger auf das letzte untergeordnete Fenster in der entsprechenden Liste des Schnittstellenobjekts zurück.

GetModule `PTModule GetModule();`

Liefert einen Zeiger auf das *TModule*-Objekt zurück, das dieses Objekt besitzt.

GetSiblingPtr `void GetSiblingPtr(Ipstream is, RPTWindowsObject P);`
protected

Liest aus dem angegebenen Stream eine Referenz auf einen Zeiger ein, der auf ein nebengeordnetes Fenster zeigt. Ein nebengeordnetes Fenster hat dasselbe übergeordnete Fenster wie this.

Beispielsweise sind *TCheckBox* und das *TGroupBox*-Objekt, das zu *TCheckBox* gehört, nebengeordnete Fenster in einem Dialogfenster.

GetSiblingPtr sollte nur während einer *read*-Operation für einen Zeiger auf ein untergeordnetes Fenster benutzt werden, der durch einen Aufruf von *PutSiblingPtr* in den Stream geschrieben wurde. Der in *P* geladene Wert wird erst gültig, wenn das übergeordnete Fenster seine *read*-Operation beendet hat. Aus diesem Grund führt die Dereferenzierung eines Zeigers auf ein nebengeordnetes Fenster innerhalb eines Stream-Konstruktors nicht zu einem korrekten Ergebnis.

Querverweis: *TWindowsObject::PutSiblingPtr, TWindowsObject::GetChildPtr, TWindowsObject::PutChildPtr*

GetWindowClass `virtual void GetWindowClass(WNDCLASS _FAR &AWndClass);`
protected

Wird von abgeleiteten Klassen überschrieben, um die angegebene Fensterklassen-Struktur mit Registrierungsattributen zu füllen.

hashValue `virtual hashValueType hashValue() const;`

Für Container-Klassen erforderlich. Überschreibt die abstrakte Funktion in der Klasse *Object*. Liefert *HWindow*, den Hash-Wert eines *TWindowsObject* zurück.

isA `virtual classType isA() const = 0;`

Abstrakte Funktion. Sie wird in abgeleiteten Klassen überschrieben, um einen eindeutigen Klassenbezeichner zurückzuliefern.

isEqual `virtual int isEqual(RCObject testwin) const;`

Für Container-Klassen erforderlich. Überschreibt die abstrakte Funktion in der Klasse *Object*. Liefert *TRUE* zurück, wenn this auf *testwin* zeigt.

isFlagSet `BOOL IsFlagSet(WORD Mask);`

Liefert den Status des Bit-Flags, das in *Mask* angegeben ist. Ist das Bit-Flag gesetzt, wird *TRUE* zurückgeliefert, andernfalls *FALSE*.

Querverweis: *TWindowsObject::SetFlags*

nameOf

```
virtual Pchar nameOf() const = 0;
```

Abstrakte Funktion. Sie wird in abgeleiteten Klassen überschrieben, um einen Klassenbezeichnungsstring zurückzuliefern.

Next

```
PTWindowsObject Next();
```

Liefert einen Zeiger auf das nächste Fenster in der Liste der untergeordneten Fenster des übergeordneten Fensters zurück. Handelt es sich dabei um das letzte Fenster in der Liste, wird ein Zeiger auf das erste Fenster zurückgeliefert.

Querverweis: *TWindowsObject::Previous*

printOn

```
virtual void printOn(Rostream outputStream) const;
```

Für Container-Klassen erforderlich. Überschreibt die abstrakte Funktion in der Klasse *Object*. Gibt eine hexadezimale Darstellung von *HWindow* in den angegebenen Stream aus.

Previous

```
PTWindowsObject Previous();
```

Liefert einen Zeiger auf das vorhergehende Fenster in der Liste der untergeordneten Fenster zurück.

Querverweis: *TWindowsObject::Next*

PutChildPtr

```
void PutChildPtr(Ropstream os, PTWindowsObject P);
```

protected

Schreibt ein untergeordnetes Fenster in den angegebenen Stream. Rufen Sie *PutChildPtr* nur während einer *write*-Operation auf, wenn Sie Zeiger auf untergeordnete Fenster in den Stream schreiben wollen, um sie später mit *GetChildPtr* einzulesen.

Querverweis: *TWindowsObject::GetSiblingPtr, TWindowsObject::PutSiblingPtr, TWindowsObject::GetChildPtr*

PutChildren

```
void PutChildren(Ropstream os);
```

Schreibt untergeordnete Fenster aus der entsprechenden Liste in den angegebenen Stream. *PutChildren* sollte nur während einer *write*-Operation aufgerufen werden, um untergeordnete Fenster in einen Stream zu schreiben, die später mit *GetChildren* eingelesen werden sollen.

PutSiblingPtr

```
void PutSiblingPtr(Ropstream os, PTWindowsObject P);
```

protected

Schreibt einen Zeiger auf ein nebengeordnetes Fenster in den angegebenen Stream. Nebengeordnete Fenster haben dasselbe übergeordnete Fenster. *PutSiblingPtr* sollte nur während einer *write*-Operation benutzt werden, um Zeiger in den Stream zu schreiben, die später durch einen Aufruf von *GetSiblingPtr* gelesen werden sollen.

Querverweis: *TWindowsObject::GetSiblingPtr, TWindowsObject::GetChildPtr, TWindowsObject::PutChildPtr*

read `virtual Pvoid read(Ripstream is);` protected

Setzt *HWindow*, *Parent*, und *TransferButton* sowie das private Datenelement *Childlist* auf 0. Erstellt eine Objektinstanz und liest *Title*, *Status*, sowie die privaten Datenelemente *Flags* und *CreateOrder* ein. Ruft *GetChildren* auf, um die untergeordneten Fenster einzulesen.

Register `virtual BOOL Register();`

Registriert die Fensterklasse von this in Windows, wenn sie noch nicht registriert ist. Der Name der Fensterklasse und die Klassenattribute werden durch den Aufruf von *GetClassName* und *GetWindowClass* ermittelt. *Register* liefert *TRUE* zurück, wenn die Fensterklasse registriert ist.

Querverweis: *TWindowsObject::GetClassName, TWindowsObject::GetWindowClass*

RemoveClient `void RemoveClient();` protected

Entfernt das angegebene Client-Fenster aus der Liste der untergeordneten Fenster. Das Client-Fenster muß unabhängig von seinen nebengeordneten Fenstern bearbeitet werden.

SetCaption `void SetCaption(LPSTR ATitle);`

Kopiert *ATitle* in einen reservierten String, auf den *Title* zeigt, und setzt den Titel des Schnittstellenelements auf *ATitle*. Die Funktion löscht einen eventuell vorhandenen Titel.

SetFlags `void SetFlags(WORD Mask, BOOL OnOff);`

Schaltet ein Bit-Flag ein oder aus (abhängig von dem in *OnOff* übergebenen Booleschen Parameter). Wenn *TRUE* angegeben ist, wird das Bit in *Mask* gesetzt. Andernfalls wird das Bit gelöscht.

Mask kann aus einer oder mehreren *WB_*-Konstanten bestehen.

Querverweis: *TWindowsObject::IsFlagSet*

SetParent `virtual void SetParent(PTWindowsObject NewParent);`

Setzt *Parent* auf das angegebene übergeordnete Fensterobjekt. Entfernt this aus der Liste der untergeordneten Fenster des vorherigen übergeordneten Fensters (falls vorhanden) und fügt this in die entsprechende Liste des neuen übergeordneten Fensters ein.

SetTransfer-Buffer `void SetTransferBuffer(Pvoid ATransferBuffer);`

Setzt *TransferBuffer* to *ATransferBuffer*.

SetupWindow `virtual void SetupWindow();` protected

Führt nach der Erstellung eines zugehörigen Schnittstellenelements den erforderlichen Aufbau durch. Iteriert über die Liste der untergeordneten Fenster und versucht, für jedes untergeordnete Fensterobjekt, für das eine automatische Erstellung festgelegt ist, das zugehörige Schnittstellenelement zu erzeugen. Die automatische Erstellung ist für Fenster und Steuerelemente per Voreinstellung aktiviert, für Dialogfenster dagegen ausgeschaltet.

Kann kein untergeordnetes Fenster erstellt werden, wird *Status* auf *EM_INVALIDCHILD* gesetzt. *SetupWindow* ruft anschließend *TransferData* auf. Diese Elementfunktion kann für abgeleitete Klassen überschrieben werden, um zusätzliche spezielle Initialisierungen durchzuführen.

Show `void Show(int ShowCmd);`

Show zeigt das Schnittstellenelement so auf dem Bildschirm an, wie es durch den Wert in *ShowCmd* angegeben ist. Die folgende Tabelle enthält die zulässigen Werte für *ShowCmd*:

Wert	Beschreibung
SW_HIDE	Verborgen
SW_SHOW	Mit der aktuellen Fenstergröße und an der aktuellen Position
SW_SHOWMAXIMIZED	In maximaler Größe und aktiv
SW_SHOWMINIMIZED	In maximaler Größe und aktiv
SW_SHOWNORMAL	Wiederhergestellt und aktiv

Diese Werte sind in windows.h definiert.

ShutDownWindow `virtual void ShutDownWindow();`

Gibt das zugehörige Schnittstellenelement ohne jede Prüfung frei und löscht this.

Transfer `virtual WORD Transfer(Pvoid DataPtr, WORD TransferFlag);`

Überträgt Fensterdaten in und aus dem angegebenen Datenpuffer. Das angegebene *TransferFlag* zeigt an, ob Daten aus dem Puffer gelesen oder in ihn geschrieben werden sollen, oder ob nur die Größe der Transferdaten zurückgeliefert werden soll. Der Rückgabewert ist der Umfang der Transferdaten in Byte. Diese Elementfunktion liefert nur 0 zurück und wird in von *TControl* abgeleiteten Klassen überschrieben.

TransferData `virtual WORD TransferData(WORD Direction);`

Ist *TransferBuffer* ungleich Null, überträgt *TransferData* Daten in oder aus dem Puffer und den beteiligten untergeordneten Fenstern des Schnittstellenobjekts. (Zwischen untergeordne-

ten Fenstern, deren *WB_TRANSFER*-Flag nicht gesetzt ist, werden keine Daten übertragen.)

TransferData ruft die *Transfer*-Elementfunktion jedes beteiligten untergeordneten Fensters auf und übergibt dabei einen Zeiger auf die Transferdaten und die in *Direction* angegebene Richtung (*TF_SETDATA*, *TF_GETDATA* oder *TF_SIZEDATA*).

Querverweis: *TWindowsObject::EnableTransfer*, *TWindowsObject::DisableTransfer*, *TWindowsObject::SetupWindow*

WMActivate
```
virtual void WMActivate(RTMessage Msg) =
[WM_FIRST + WM_ACTIVATE];
```
protected

Antwortet auf eine eingehende *WM_ACTIVATE*-Botschaft.

Aktiviert oder deaktiviert die Tastaturbehandlung und ruft dazu *ActivationResponse* auf.

Querverweis: *TWindowsObject::ActivationResponse*

WMClose
```
virtual void WMClose(RTMessage Msg) =
[WM_FIRST + WM_CLOSE];
```
protected

Antwortet mit einem Aufruf von *CloseWindow* auf die Anforderung, das Fenster zu schließen.

Querverweis: *TWindowsObject::Destroy*

WMCommand
```
virtual void WMCommand(RTMessage Msg) =
[WM_FIRST + WM_COMMAND];
```
protected

Verteilt Botschaften, die auf Befehlen, auf der ID untergeordneter Fenster oder auf Benachrichtigungen basieren, und ruft die entsprechende Botschaftsantwortfunktion auf.

WMDestroy
```
virtual void WMDestroy(RTMessage Msg) =
[WM_FIRST + WM_DESTROY];
```
protected

Beantwortet eine eingehende *WM_DESTROY*-Botschaft. Ist this das Hauptfenster, sendet *WMDestroy* eine »Quit«-Botschaft, um die Anwendung zu beenden.

WMDrawItem
```
virtual void WMDrawItem(RTMessage Msg) =
[WM_FIRST + WM_DRAWITEM];
```
protected

Verteilt *WM_DRAWITEM*-Botschaften für benutzergezeichnete Steuerelemente durch den Aufruf der *WMDrawItem*-Elementfunktion des Steuerelements. Ruft *WMDrawItem* auf, wenn die *WM_DRAWITEM*-Botschaft an einen benutzergezeichneten Menüeintrag gesendet wird.

Querverweis: *TWindowsObject::DrawItem*, *WMTControl::DrawItem*

WMHScroll
```
virtual void WMHScroll(RTMessage Msg) =
[WM_FIRST + WM_HSCROLL];
```
protected

Beantwortet eine eingehende *WM_HSCROLL* -Botschaft durch einen Aufruf von *DispatchScroll*.

Querverweis: *TWindowsObject::DispatchScroll*

WMNCDestroy `virtual void WMNCDestroy(RTMessage Msg) = [WM_FIRST + WM_NCDESTROY];` protected

Beantwortet eine eingehende *WM_NCDESTROY* -Botschaft, die als letzte Botschaft an ein Windows-Schnittstellenelement gesendet wird. Setzt *HWindow* auf 0, um anzuzeigen, daß ein Schnittstellenelement nicht mehr mit this verknüpft ist. Löscht this, wenn es für ein Nicht-ObjectWindows-Fenster steht.

WMQueryEndSession `virtual void WMQueryEndSession(RTMessage Msg) = [WM_FIRST + WM_QUERYENDSESSION];` protected

Antwortet auf den Versuch von Windows, die Sitzung zu beenden. Ist this das Hauptfenster, liefert *WMQueryEndSession* das Ergebnis eines *GetApplication* ()->*CanClose*-Aufrufs zurück. Andernfalls wird das Ergebnis des Aufrufs *this->CanClose* zurückgeliefert.

WMVScroll `virtual void WMVScroll(RTMessage Msg) = [WM_FIRST + WM_VSCROLL];` protected

Beantwortet eine eingehende *WM_VSCROLL* -Botschaft durch einen Aufruf von *DispatchScroll*.

Querverweis: *TWindowsObject::DispatchScroll*

write `virtual void write(Ropstream os);` protected

Schreibt *Title, Status* sowie die privaten Datenelemente *Flags* und *CreateOrder* in den Stream. *PutChildren* wird aufgerufen, um untergeordnete Fenster an den Stream zu übergeben.

23. Stream-fähige Klassen in ObjectWindows

Im vorliegenden Kapitel finden Sie eine alphabetische Auflistung aller öffentlichen (public) Klassen, die stream-fähige Objekte unterstützen. Dabei werden Verwendung, Elemente, Operatoren und *Friends* der Klassen erläutert.

Wie Sie am schnellsten zur gewünschten Information gelangen, haben wir am Beginn des Kapitels 22 »ObjectWindows-Klassen« erklärt.

Die Hierarchie der Streams

Die Stream-Bibliothek von ObjectWindows besteht aus einer Hierarchie von Klassen, die mit den stream-fähigen Objekten verknüpft sind. Das vorliegende Kapitel ist eine Referenz auf diese Klassen. Sehen Sie sich dazu Abbildung 23.1 an. Die Klasse *TStreamable* in diesem Diagramm ist eine *Friend*-Klasse.

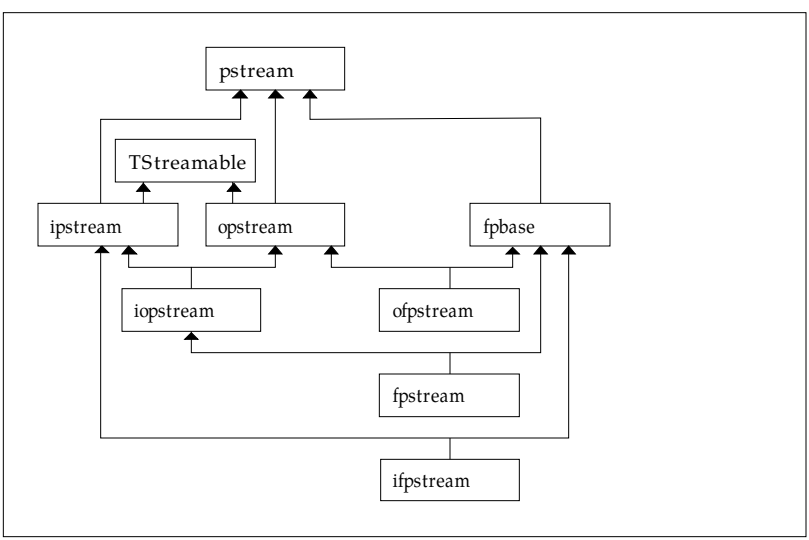
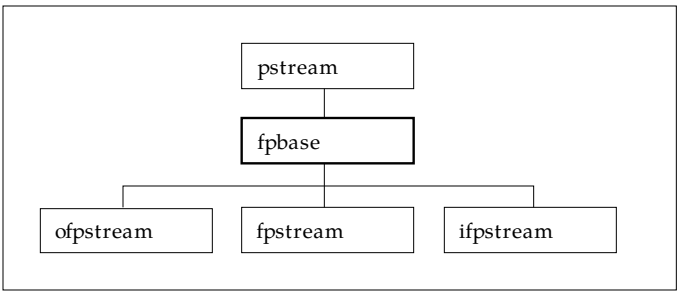


Abbildung 23.1 Streamklassen-Hierarchie in ObjectWindows.

fpbase

objstrm.h



fpbase stellt die grundlegenden Ein-/Ausgabeoperationen zur Verfügung, die allen Datei-Streams gemeinsam sind.

Element-funktionen

Konstruktor

```
fpbase();
fpbase(PCchar name, int omode, int prot =
       filebuf::openprot);
fpbase(int f);
fpbase(int f, Pchar b, int len);
```

Erstellt ein gepuffertes *fpbase*-Objekt. Die Größe und der anfängliche Inhalt des Puffers können in den Argumenten *len* und *b* gesetzt werden. Entweder es wird eine Datei geöffnet (dazu übergibt man den Namen, Modus und Zugriffsrechte) oder man übergibt eine bereits geöffnete Datei im Datei-Deskriptor *f*.

Wenn Sie keine Datei angeben, müssen Sie das später mit *open* nachholen.

Destruktor

```
~fpbase();
```

Gibt das *fpbase*-Objekt frei.

attach

```
void attach(int f);
```

Ordnet die Datei mit dem Deskriptor *f* diesem Stream zu (falls möglich), und setzt *ios::state* entsprechend.

close

```
void close();
```

Schließt den Stream und die zugehörige Datei.

open

```
void open(PCchar name, int mode, int prot =
          filebuf::openprot);
```

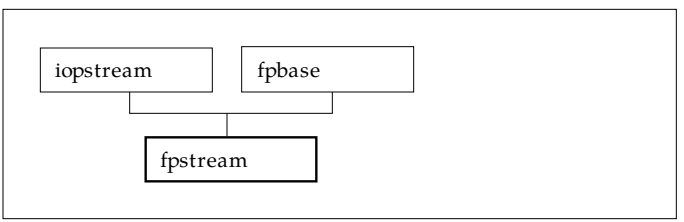
Öffnet die angegebene Datei mit gegebenem Modus (*app, ate, in, out, binary, trunc, nocreate, noreplace*) und Zugriffsrechten. Die geöffnete Datei wird mit diesem Stream verknüpft.

rdbuf
```
Pfilebuf rdbuf();
```
Liefert einen Zeiger auf den aktuellen Dateipuffer.

setbuf
```
void setbuf(Pchar buf, int len);
```
Reserviert Speicherplatz für einen Puffer mit der Größe *len*.

fpstream

objstrm.h



fpstream kombiniert einfach die Basisklassen *fpbase* und *iopstream*. Sie bildet die Basisklasse für das gleichzeitige Lesen und Schreiben stream-fähiger Objekte in bidirektionalen Datei-Streams.

Element-funktionen

Konstruktor
```
fpstream();
fpstream(PCchar name, int mode, int prot =
        filebuf::openprot);
fpstream(int f);
fpstream(int f, Pchar b, int len);
```
Erstellt ein gepuffertes *fpstream*-Objekt. Die Größe und der anfängliche Inhalt des Puffers können in den Argumenten *len* und *b* gesetzt werden. Entweder es wird eine Datei geöffnet (dazu übergibt man den Namen, Modus und Zugriffsrechte) oder man übergibt eine bereits geöffnete Datei im Datei-Deskriptor *f*.

Wenn Sie keine Datei angeben, müssen Sie das später mit *open* nachholen.

Destruktor
```
~fpstream();
```
Gibt das *fpstream*-Objekt frei.

open
```
void open(PCchar name, int mode, int prot =
         filebuf::openprot);
```
Öffnet die angegebene Datei mit gegebenem Modus (*app, ate, in, out, binary, trunc, nocreate, noreplace*) und Zugriffsrechten. Die geöffnete Datei wird mit diesem Stream verknüpft.

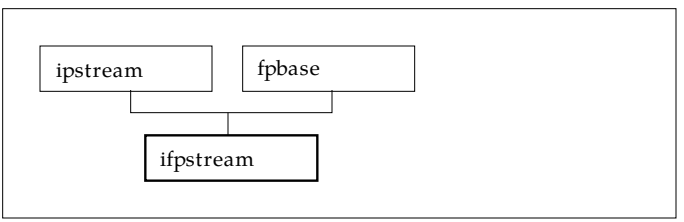
rdbuf

```
Pfilebuf rdbuf();
```

Liefert einen Zeiger auf den aktuellen Dateipuffer.

ifpstream

objstrm.h



ifpstream kombiniert einfach die Basisklassen *fpbase* und *ipstream*. Sie bildet die Basisklasse zum Lesen stream-fähiger Objekte aus Datei-Streams.

Element-funktionen

Konstruktor

```
ifpstream();
ifpstream(PCchar name, int mode = ios::in,
          int prot = filebuf::openprot);
ifpstream(int f);
ifpstream(int f, Pchar b, int len);
```

Erstellt ein gepuffertes *ifpstream*-Objekt. Die Größe und der anfängliche Inhalt des Puffers können in den Argumenten *len* und *b* gesetzt werden. Entweder es wird eine Datei geöffnet (dazu übergibt man den Namen, Modus und Zugriffsrechte) oder man übergibt eine bereits geöffnete Datei im Datei-Deskriptor *f*.

Wenn Sie keine Datei angeben, müssen Sie das später mit *open* nachholen.

Destruktor

```
~ifpstream();
```

Gibt das *ifpstream*-Objekt frei.

open

```
void open(PCchar name, int mode = ios::in,
          int prot = filebuf::openprot);
```

Öffnet die angegebene Datei mit gegebenem Modus (*app, ate, in, out, binary, trunc, nocreate, noreplace*) und Zugriffsrechten. Voreingestellt ist der Modus *in* (Eingabe) mit den Zugriffsrechten von *openprot*. Die geöffnete Datei wird mit diesem Stream verknüpft.

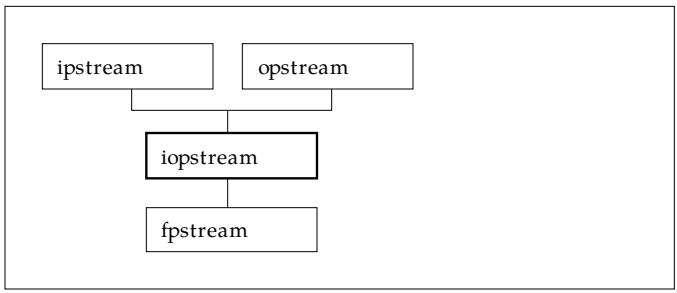
rdbuf

```
Pfilebuf rdbuf();
```

Liefert einen Zeiger auf den aktuellen Dateipuffer.

iopstream

objstrm.h



iopstream kombiniert die Basisklassen *opstream* und *ipstream*. Sie stellt die Basisklasse zum gleichzeitigen Lesen und Schreiben stream-fähiger Objekte zur Verfügung.

Element-funktionen

Konstruktor

```
iopstream(Pstreambuf buf);
iopstream();                                    protected
```

Die erste Form des Konstruktors erstellt ein gepuffertes *iopstream*-Objekt mit dem gegebenen Puffer und setzt das Datenelement *bp* auf *buf*. Der Status wird auf 0 gesetzt.

Die zweite Form initialisiert den Pufferzeiger *bp* nicht. Verwenden Sie *init* für das Setzen des Puffers und des Status.

Querverweis: *pstream::init*

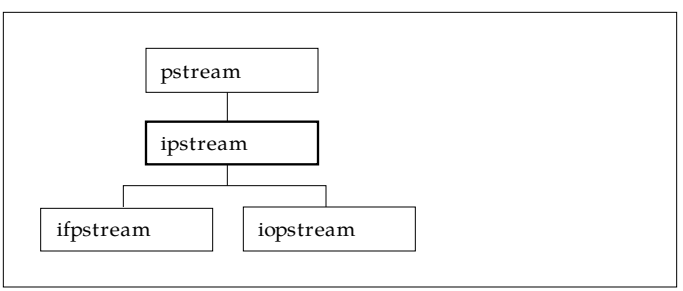
Destruktor

```
~iopstream();
```

Gibt das *iopstream*-Objekt frei.

ipstream

objstrm.h



ipstream, ein von *pstream* abgeleiteter, spezialisierter Eingabe-Stream, dient als Basisklasse für das Einlesen stream-fähiger Objekte aus einem Stream.

Element-funktionen

Konstruktor
```
ipstream(Pstreambuf);
ipstream();                                    protected
```
Die erste Form des Konstruktors erstellt ein gepuffertes *ipstream*-Objekt mit dem gegebenen Puffer und setzt das Datenelement *bp* auf *buf*. Der Status wird auf 0 gesetzt. Die zweite Form initialisiert den Pufferzeiger *bp* nicht. Verwenden Sie *init* für das Setzen des Puffers und des Status.

Querverweis: *pstream::init*

Destruktor
```
~ipstream();
```
Gibt das *ipstream*-Objekt frei.

find
```
PCvoid find(P_id_type Id);                     protected
```
Liefert einen Zeiger auf das *Id* entsprechende Objekt zurück.

freadBytes
```
void freadBytes(void far *data, size_t sz);
```
Liest die Anzahl der von *sz* angegebenen Bytes in den gegebenen far-Puffer (*data*).

freadString
```
char far *freadString();
```
Liest einen String aus dem Stream. Ermittelt die Länge des Strings und reserviert in der entsprechenden Größe Speicherplatz für ein Zeichen-Array vom Typ far, liest den String in dieses Array und liefert einen Zeiger auf den String zurück. Der Aufrufer sollte den reservierten Speicherblock freigeben.

freadString
```
char far *freadString(char far *buf, unsigned maxLen);
```
Liest einen String aus dem Stream in den angegebenen far-Puffer (*buf*). Wenn die Länge des Strings größer als *maxLen*-1 ist, werden nur die ersten *maxLen*-1 Zeichen gelesen. Der String wird mit 0 abgeschlossen.

readByte
```
uchar readByte();
```
Liefert das Zeichen an der aktuellen Stream-Position zurück.

readBytes
```
void readBytes(Pvoid data, size_t sz);
```
Liest *sz* Bytes von der aktuellen Stream-Position ein und schreibt sie nach *data*.

readData
```
Pvoid readData(PCTStreamableClass,
PCTStreamable mem);                            protected
```
Liest die Daten aus dem Stream in das Objekt, auf das *mem* zeigt und benutzt dafür die *read*-Funktion des Objekts. Wenn *mem* gleich Null ist, wird zuerst die entsprechende *build*-Funktion aufgerufen.

Querverweis: *TStreamableClass*, *read*- und *build*-Elementfunktionen jeder stream-fähigen Klasse.

readPrefix `PCTStreamableClass readPrefix();` protected

Liefert das *TStreamableClass*-Objekt zurück, das der Klasse mit dem Namen *name* entspricht, der an der aktuellen Position gespeichert ist.

readString `Pchar readString();`

Reserviert Speicherplatz für einen Puffer, der groß genug für die Aufnahme des Strings an der aktuellen Stream-Position ist. Liest den String aus dem Stream in den Puffer. Der Aufrufer muß den Puffer freigeben.

readString `Pchar readString(Pchar buf, unsigned maxLen);`

Liest den String an der aktuellen Stream-Position in den von *buf* angegebenen Puffer. Ist der String größer als *maxLen*-1, werden nur die ersten *maxLen*-1 Byte kopiert.

readSuffix `void readSuffix();` protected

Liest und überprüft das letzte Byte im Namensfeld eines Objekts.

Querverweis: *ipstream::readPrefix*

readWord `ushort readWord();`

Liefert das *WORD* an der aktuellen Stream-Position zurück.

registerObject `void registerObject(PCvoid adr);` protected

Registriert die Klasse des Objekts, auf das *adr* zeigt.

seekg
```
Ripstream seekg(streampos pos);
Ripstream seekg(streamoff off, seek_dir dir);
```

Die erste Version verschiebt die Stream-Position auf die absolute, in *pos* gegebene Position. Die zweite Version verschiebt die Stream-Position relativ zur in *dir* angegebenen Position um einen Offset von *off* (+ oder -). *dir* kann auf *beg* (Beginn des Streams), *cur* (aktuelle Stream-Position) oder *end* (Ende des Streams) gesetzt sein.

tellg `streampos tellg();`

Liefert die (absolute) aktuelle Stream-Position.

Friends

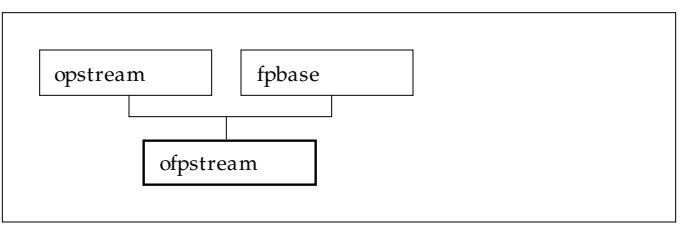
Operator >>
```
friend Ripstream operator >> (Ripstream ps,
                              signed char _FAR & ch);
friend Ripstream operator >> (Ripstream ps,
                              unsigned char _FAR & ch);
```

```
friend Ripstream operator >> (Ripstream ps,
                              signed short _FAR & sh);
friend Ripstream operator >> (Ripstream ps,
                              unsigned short _FAR & sh);
friend Ripstream operator >> (Ripstream ps,
                              signed int _FAR & i);
friend Ripstream operator >> (Ripstream ps,
                              unsigned int _FAR & i);
friend Ripstream operator >> (Ripstream ps,
                              signed long _FAR & l);
friend Ripstream operator >> (Ripstream ps,
                              unsigned long _FAR & l);
friend Ripstream operator >> (Ripstream ps,
                              float_FAR & f);
friend Ripstream operator >> (Ripstream ps,
                              double _FAR & d);
friend Ripstream operator >> (Ripstream ps,
                              long double _FAR & d);
friend Ripstream operator >> (Ripstream ps,
                              RTStreamable t);
friend Ripstream operator >> (Ripstream ps, RPvoid t);
```

Liest aus dem *ipstream ps* in das gegebene Argument. Es wird eine Referenz auf den Stream zurückgeliefert, wodurch Sie die >>Operatoren wie üblich verketten können.

ofpstream

objstrm.h



ofpstream kombiniert die Basisklassen *fpbase* und *opstream*. Sie stellt die Basisklasse für das Schreiben stream-fähiger Objekte in Datei-Streams.

Element-funktionen

Konstruktor

```
ofpstream();
ofpstream(PCchar name, int mode = ios::out,
          int prot = filebuf::openprot);
ofpstream(int f);
ofpstream(int f, Pchar b, int len);
```

Erstellt ein gepuffertes *ofpstream*-Objekt. Die Größe und der anfängliche Inhalt des Puffers können in den Argumenten *len* und *b* gesetzt werden. Entweder es wird eine Datei geöffnet (dazu übergibt man den Namen, Modus und Zugriffsrechte) oder man übergibt eine bereits geöffnete Datei im Datei-Deskriptor *f*. Wenn Sie keine Datei angeben, müssen Sie das später mit *open* nachholen.

Destruktor

```
~ofpstream();
```

Gibt das *ofpstream*-Objekt frei.

open

```
void open(PCchar name, int mode = ios::out,
          int prot = filebuf::openprot);
```

Öffnet die angegebene Datei mit gegebenem Modus (*app, ate, in, out, binary, trunc, nocreate* oder *noreplace*) und Zugriffsrechten. Voreingestellt ist der Modus *out* (Ausgabe) mit den Zugriffsrechten *openprot*. Die geöffnete Datei wird mit diesem Stream verknüpft.

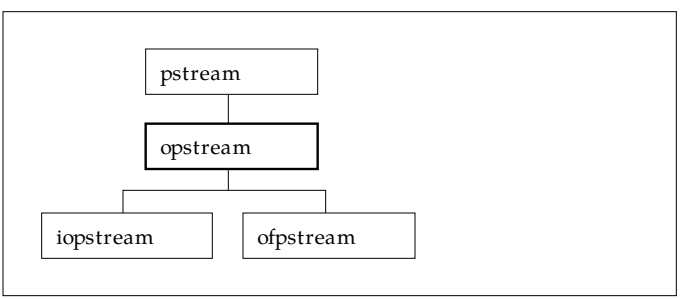
rdbuf

```
Pfilebuf rdbuf();
```

Liefert den aktuellen Dateipuffer.

opstream

objstrm.h



opstream, ein von *pstream* abgeleiteter, spezialisierter Eingabe-Stream, dient als Basisklasse für das Schreiben von streamfähigen Objekten in einen Stream.

Element funktionen

Konstruktor

```
opstream(Pstreambuf buf);
opstream();                                    protected
```

Die erste Form des Konstruktors erstellt ein gepuffertes *opstream*-Objekt mit dem gegebenen Puffer und setzt das Datenelement *bp* auf *buf*. Der Status wird auf 0 gesetzt. Die zweite Form initialisiert den Pufferzeiger *bp* nicht. Verwenden Sie *init* für das Setzen des Puffers und des Status.

Destruktor `~opstream();`

Gibt das *opstream*-Objekt frei.

find `P_id_type find(PCvoid adr);` protected

Liefert die Typ-ID des Objekts zurück, auf das *adr* zeigt.

flush Ropstream flush();

Entleert den Puffer in den Stream.

fwriteBytes `void fwriteBytes(const void far *data, size_t sz);`

Überträgt die (in *sz*) angegebene Anzahl Byte aus dem gegebenen far-Puffer (*data*) in den Stream.

fwriteString `void fwriteString(const char far *str);`

Schreibt den angegebenen far-String (*str*) in den Stream.

registerObject `void registerObject(PCvoid adr);` protected

Registriert die Klasse des Objekts, auf das *adr* zeigt.

seekp
```
Ropstream seekp(streampos pos);
Ropstream seekp(streamoff off, seek_dir dir);
```

Die erste Form verschiebt die aktuelle Stream-Position auf die absolute, in *pos* gegebene Position. Die zweite Zeile verschiebt die Stream-Position relativ zur in *dir* angegebenen Position um einen Offset von *off* (+ oder -). *dir* kann auf *beg* (Beginn des Streams), *cur* (aktuelle Stream- Position) oder *end* (Ende des Streams) gesetzt sein.

tellp `streampos tellp();`

Liefert die (absolute) aktuelle Stream-Position.

writeByte `void writeByte(uchar ch);`

Schreibt das Byte *ch* in den Stream.

writeBytes `void writeBytes(PCvoid data, size_t sz);`

Schreibt *sz* Byte aus dem *data*-Puffer in den Stream.

writeData `void writeData(RTStreamable);` protected

Schreibt Daten in den Stream und ruft dazu die Elementfunktion *write* des zu schreibenden Objekts auf.

Querverweis: *TStreamable*, *write*-Funktionen in den streamfähigen Klassen.

writePrefix `void writePrefix(RCTStreamable);` protected

Schreibt das Klassennamen-Präfix in den Stream. Der <<-Operator umgibt die mit *writeData* in den Stream geschriebenen Daten mit einem Präfix und einem Suffix.

Präfix und Suffix werden für typensichere Stream-Ein- und -Ausgaben benötigt.

Querverweis: *ipstream:readPrefix*

writeString `void writeString(PCchar str);`

Schreibt *str* (zusammen mit einem führenden Länge-Byte) in den Stream.

writeSuffix `void writeSuffix(RCTStreamable);` protected

Schreibt das Klassennamen-Suffix in den Stream. Der <<-Operator umgibt die mit *writeData* in den Stream geschriebenen Daten mit einem Präfix und einem Suffix.

Präfix und Suffix werden für typensichere Stream-Ein- und -Ausgaben benötigt.

Querverweis: *ipstream:readSuffix*

writeWord `void writeWord(ushort us);`

Schreibt das *WORD us* in den Stream.

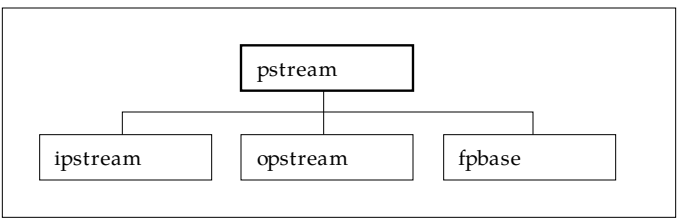
Friends

Operator <<

```
friend Ropstream operator << (Ropstream ps,
                              signed char ch);
friend Ropstream operator << (Ropstream ps,
                              unsigned char ch);
friend Ropstream operator << (Ropstream ps,
                              signed short sh);
friend Ropstream operator << (Ropstream ps,
                              unsigned short sh);
friend Ropstream operator << (Ropstream ps,
                              signed int i);
friend Ropstream operator << (Ropstream ps,
                              unsigned int i);
friend Ropstream operator << (Ropstream ps,
                              signed long l);
friend Ropstream operator << (Ropstream ps,
                              unsigned long l);
friend Ropstream operator << (Ropstream ps, float f);
friend Ropstream operator << (Ropstream ps, double d);
friend Ropstream operator << (Ropstream ps,
                              long double d);
friend Ropstream operator << (Ropstream ps,
                              RTStreamable t);
friend Ropstream operator << (Ropstream ps,
                              PTStreamable t );
```

Fügt das gegebene Argument in das *ipstream*-Objekt ein.

pstream objstrm.h



pstream ist die Basisklasse für die Verwaltung stream-fähiger Objekte.

Datenelemente

bp `Pstreambuf bp;` protected

Ein Zeiger auf den Stream-Puffer.

state `int state;` protected

Flags für den Formatstatus. Verwenden Sie für den Zugriff auf den aktuellen Status *rdstate*.

Querverweis: *pstream::rdstate*

types `static PTStreamableTypes types;` protected

Ein Zeiger auf eine Datenbank, die alle registrierten Typen der Anwendung enthält.

Querverweis: *pstream::InitTypes*

Element-funktionen

Konstruktor
```
pstream(Pstreambuf buf);
pstream();
```
protected

Die erste Form des Konstruktors erstellt ein gepuffertes *pstream*-Objekt mit dem gegebenen Puffer und setzt das Datenelement *bp* auf *buf*. Der Status wird auf 0 gesetzt.

Die zweite Form reserviert einen Standardpuffer.

Destruktor `virtual ~pstream();`

Gibt das *pstream*-Objekt frei.

bad `int bad() const;`

Liefert im Falle eines Fehlers einen Wert zurück, der ungleich 0 ist.

clear `void clear(int aState = 0);`

Setzt den Stream-Status (*state*) auf den gegebenen Wert (voreingestellt ist 0).

eof

```
int eof() const;
```

Liefert am Ende des Streams einen Wert zurück, der ungleich 0 ist.

error

```
void error(StreamableError, RCTStreamable);
void error(StreamableError);
```

protected

Setzt die gegebene Fehlerbedingung, wobei *StreamableError* folgendermaßen definiert ist:

```
enum StreamableError { peNotRegistered, peInvalidType };
```

fail

```
int fail() const;
```

Liefert einen Wert ungleich 0 zurück, wenn eine Stream-Operation fehlschlägt.

good

```
int good() const;
```

Liefert einen Wert ungleich 0 zurück, wenn keine Status-Bits gesetzt sind (d.h. wenn keine Fehler aufgetreten sind).

init

```
void init(Pstreambuf sbp);
```

protected

Initialisiert den Stream: setzt *state* auf 0 und *bp* auf *sbp*.

initTypes

```
static void initTypes();
```

Erstellt das Datenbank-Objekt, auf das *types* zeigt. Wird vom *TStreamableClass*-Konstruktor aufgerufen.

Querverweis: *TStreamableClass*

operator void *()

```
operator Pvoid() const;
```

Überschreibt den Typumwandlungsoperator Zeiger-auf-*void*.

Bei fehlgeschlagener Ausführung liefert die Elementfunktion 0 zurück (d.h., wenn *pstream::fail* einen Wert ungleich 0 zurückgibt), andernfalls wird ein Wert zurückgeliefert, der ungleich 0 ist.

Querverweis: *pstream::fail*

operator ! ()

```
int operator ! () const;
```

Überschreibt den Negations-Operator (!). Liefert den von *pstream::fail* zurückgegebenen Wert zurück.

Querverweis: *pstream::fail*

rdbuf

```
Pstreambuf rdbuf() const;
```

Liefert den Zeiger *pb* auf den mit dem Stream verknüpften Puffer zurück.

Querverweis: *pstream::pb*

rdstate

```
int rdstate() const;
```

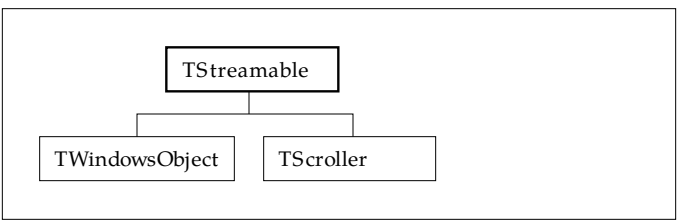
Liefert den aktuellen *state*-Wert zurück.

setstate `void setstate(int b);` protected

Aktualisiert das Datenelement *state* mit state |= (b & 0xFF).

Friends Die private Datenbank-Klasse ist ein *Friend* von *pstream*.

TStreamable objstrm.h



Klassen, die von *TStreamable* abgeleitet sind, werden als stream-fähige Klassen bezeichnet, da ihre Objekte in Streams geschrieben und aus Streams gelesen werden können. Bei der Entwicklung eigener stream-fähiger Klassen müssen Sie darauf achten, daß *TStreamable* als Basisklasse vorkommt. Am sinnvollsten ist es, eine vorhandene stream-fähige Klasse als Basis zu verwenden. Wenn Ihre Klasse in eine vorhandene Klassenhierarchie eingegliedert werden soll, können Sie auch die Mehrfachvererbung verwenden.

Da *TStreamable* eine abstrakte Klasse ist, können keine Objekte dieser Klasse instantiiert werden. Klassen, die von *TStreamable* erben, müssen ihre drei abstrakten Funktionen *streamableName*, *read* und *write* überschreiben.

Element-funktionen

read `virtual Pvoid read(Ripstream) = 0;` protected

Diese abstrakte Elementfunktion muß von jeder abgeleiteten Klasse überschrieben (oder als abstrakt neu deklariert) werden. Die überschreibende *read*-Funktion für jede stream-fähige Klasse muß die erforderlichen Datenelemente aus dem *ipstream*-Objekt einlesen. *read* wird normalerweise durch einen Aufruf der Elementfunktion *read* der Basisklasse (falls vorhanden) implementiert. Danach werden Datenelemente der abgeleiteten Klasse eingelesen.

Querverweis: *TStreamableClass*, *ipstream*

streamable-Name `virtual const Pchar streamableName() const = 0;` private

Diese abstrakte Elementfunktion muß von jeder abgeleiteten Klasse überschrieben (oder als abstrakt neu deklariert) werden. Diese Elementfunktion liefert den Namen der stream-fähigen

Klasse des Objekts zurück, das sie aufruft. Die Stream-Verwaltung verwendet diesen Namen bei der Registrierung von Klassen. Der zurückgelieferte Name muß ein eindeutiger, nullterminierter String sein (z.B. "TDialog")

Querverweis: *TStreamableClass*, *opstream*, *ipstream*

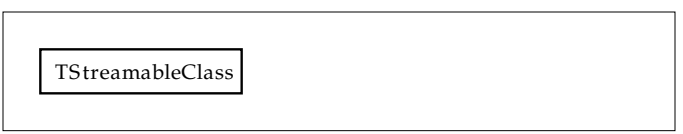
write `virtual void write(Ropstream) = 0;` protected

Diese abstrakte Elementfunktion muß von jeder abgeleiteten Klasse überschrieben (oder als abstrakt neu deklariert) werden. Die überschreibende *write*-Funktion für jede stream-fähige Klasse muß die erforderlichen Datenelemente in das *opstream*-Objekt schreiben. *write* wird normalerweise durch einen Aufruf der Elementfunktion *write* der Basisklasse (falls vorhanden) implementiert. Danach werden Datenelemente der abgeleiteten Klasse eingefügt.

Querverweis: *TStreamableClass, opstream*

Friends Die Klassen *opstream* und *ipstream* sind Friends von *TStreamable*.

TStreamableClass objstrm.h



TStreamableClass wird von der privaten Datenbank-Klasse und *pstream* bei der Registrierung stream-fähiger Klassen verwendet.

Element-funktion

Konstruktor `TStreamableClass(PCchar n, BUILDER b, int d);`

Erstellt ein *TStreamble*-Objekt mit dem gegebenen Namen (*n*) und der gegebenen *build*-Funktion (*b*) und registriert dann den Typ. Jede stream-fähige Klasse, z.B. mit dem Namen *TClassname*, verfügt über eine *build*-Elementfunktion vom Typ *BUILDER* (vgl. Kapitel 24 »Weitere Komponenten von ObjectWindows«). Für typensichere Objektstream-Ein-/Ausgabe muß die Stream-Verwaltung auf die Namen und die Typ-Informationen jeder Klasse zugreifen können. Damit bei Verwendung der Stream-Verwaltung die richtigen Funktionen zur Anwendung gelinkt werden, müssen Sie eine Referenz angeben, wie beispielsweise:

```
TStreamableClass RegClassName;
```

Dabei ist *TClassName* der Name der Klasse, in die Objekte geschrieben oder aus der Objekte gelesen werden sollen. (Beachten Sie, daß *RegClassName* ein einziger Bezeichner ist.) Dadurch wird nicht nur *TClassName* registriert, indem der Stream-Verwaltung mitgeteilt wird, welche *build*-Funktion verwendet werden soll, sondern auch automatisch alle Klassen, von denen *TClassName* abhängt. Sie können eine Klasse mehrmals registrieren, ohne Fehler zu riskieren oder Overhead in Kauf nehmen zu müssen.

Die Aufgabe der *build*-Funktion besteht darin, Speicher zu reservieren und teilweise zu initialisieren, so daß anschließend ein Objekt eingelesen werden kann. Die *build*-Funktion ruft einen speziellen Konstruktor auf, der dafür sorgt, daß Zeiger auf Tabellen virtueller Funktionen korrekt initialisiert werden.

Der Parameter *d* bezeichnet den Abstand zwischen der »Startadresse« eines Objekts und seiner *TStreamable*-Komponente. Das Makro *__DELTA* berechnet diesen Abstand (vgl. dazu Kapitel 24 »Weitere Komponenten von ObjectWindows«). Ein Beispiel:

```
TStreamableClass RegTClassName = TStreamableClass
("TClassName", TClassName::build, __DELTA(TClassName));
```

Querverweis: *TStreamable*, *ipstream*, *opstream*

Friends

Die Klassen *opstream* und *ipstream* sind Friends von *TStreamableClass*.

24. Weitere Komponenten von ObjectWindows

Das vorliegende Kapitel befaßt sich mit verschiedenen ObjectWindows-Komponenten, die in Kapitel 22 »ObjectWindows-Klassen« und 23 »Streamfähige ObjectWindows-Klassen« beschrieben sind.

Dazu gehören Typen, Konstanten, Variablen, Makros, Klassen und Funktionen, die häufig in den Header-Dateien von ObjectWindows deklariert sind.

Ein Beispieleintrag sieht folgendermaßen aus:

Beispiel

Header-Datei von Beispiel

Deklaration `RückgabeTyp (ParameterTyp AParam);`

Beispiel führt eine Funktion mit dem Parameter *AParam* aus.

Querverweis *Beispiel2*

applicationClass-Konstante

owldefs.h

Beschreibung Ein Klassenbezeichner zur Identifizierung von Instanzen der Klasse *TApplication*. Diese Konstante wird von der Elementfunktion *isA* zurückgeliefert, um für die Container-Klassen den Objekttyp zu bezeichnen.

Querverweis *TApplication::isA*

BF_XXXX-Konstanten

owldefs.h

Beschreibung Folgende Schalter-Flag-Konstanten sind definiert:

Tabelle 24.1 Schalter-Flag-Konstanten

24 **Beispiel** 427

BF_UNCHECKED	0x0	Element ist nicht eingeschaltet.
BF_CHECKED	0x1	Element ist eingeschaltet.
BF_GRAYED	0x2	Element ist deaktiviert (»grau«).

Auswahlfeld- und Schaltfeldobjekte definieren mit Hilfe der *BF_*-Konstanten ihren Zustand.

Querverweis: *TCheckBox::SetCheck*, *TCheckBox::GetCheck*

BUILDER-Typ

objstrm.h

Definition

```
typedef PTStreamable (_FAR * BUILDER)();
```

Wird als Typ für den *build*-Funktionsparameter des *TStreamableClass*-Konstruktors verwendet. Die *build*-Funktion reserviert beim Einlesen eines Objekts aus einem Stream Speicher für dieses Objekt und initialisiert es teilweise.

Querverweis: *TStreamableClass*

_CLASSDEF(Klassenname)-Makro

_defs.h

Beschreibung *_CLASSDEF* wird am Beginn der Header-Dateien verwendet, die die Klassen definieren. Das Makro definiert die folgenden Typen für die Klasse *classname*:

```
typedef classname _FAR * Pclassname;
typedef classname _FAR & Rclassname;
typedef classname _FAR * _FAR & RPclassname;
typedef const classname _FAR * PCclassname;
typedef const classname _FAR & RCclassname;
```

Diese Typdefinitionen unterstützen die Programmierung von dynamischen Linkbibliotheken. Das Makro _FAR ergibt, wenn erforderlich, das Schlüsselwort far, so daß Sie far nicht explizit für Datenzeiger und Referenzen verwenden müssen.

_CLASSDLL-Makro

Beschreibung Dieses Makro wird vom Anwender in der Compiler-Kommandozeile (oder in der IDE) gesetzt. Dadurch wird gekennzeichnet, daß das Modul Klassen in einer dynamischen Linkbibliothek verwendet oder daß das Modul Klassen an eine dynamische Linkbibliothek übergibt. Weitere Makros (vor allem *_CLASSTYPE*, *_EXPORT* und *_FAR*) überprüfen, ob *_CLASSDLL* definiert wurde.

_CLASSTYPE-Makro

_defs.h

Beschreibung Dieses Makro wird in der Header-Datei einer Klasse verwendet, die mit dynamischen Linkbibliotheken verwendet werden kann. Es erweitert entsprechend zu huge, far oder near.

Wenn die Makros *__DLL__* oder *_CLASSDLL* definiert sind, erweitert *_CLASSTYPE* zu huge. Klassen von Typ huge sind erforderlich, wenn sie zwischen einer ausführbaren Datei und einer dynamischen Linkbibliothek oder zwischen zwei dynamischen Linkbibliotheken ausgetauscht werden. Wenn die Makros *__DLL__* oder *_CLASSDLL* nicht definiert sind, erweitert *_CLASSTYPE* folgendermaßen:

Tabelle 24.2 _CLASSTYPE-Makro-Erweiterung (nicht DLL)

24 **CM_XXXX-Konstanten** 429

tiny, small, medium	near	*near* this *near vtable*
large, compact	far	*far* this *near vtable*
huge	huge	*far* this *far vtable*

CM_XXXX-Konstanten

owldefs.h

Beschreibung

Die folgenden Konstanten definieren den Anfangswert oder die Anzahl von anwenderdefinierten und internen Befehlsbotschaften:

Tabelle 24.3 Konstanten für Befehlsbotschaften

Konstante	Wert	Beschreibung
CM_COUNT	0x6000	Anzahl der anwenderdefinierten Befehlsbotschaften.
CM_FIRST	0xA000	Beginn der anwenderdefinierten Befehlsbotschaften.
CM_INTERNAL	0xFF00	Beginn der reservierten internen Befehlsbotschaften.
CM_RESERVED		*CM_INTERNAL - CM_FIRST*

Die folgende Tabelle enthält die Konstantenbezeichner und Klassen mit Elementfunktionen, die diese Befehlsbotschaften definieren:

Tabelle 24.4 Befehlskonstanten

Konstante	Name
CM_EDITCUT	TEdit::CMEditCut *Menü-Äquivalent:* Edit/Cut
CM_EDITCOPY	TEdit::CMEditCopy *Menü-Äquivalent:* Edit/Copy

430 **Weitere Komponenten von ObjectW indows** 24

CM_EDITPASTE	TEdit::CMEditPaste *Menü-Äquivalent:* Edit/Paste
CM_EDITDELETE	TEdit::CMEditDelete *Menü-Äquivalent:* Edit/Delete
CM_EDITCLEAR	TEdit::CMEditClear *Menü-Äquivalent:* Edit/Clear
CM_EDITUNDO	TEdit::CMEditUndo *Menü-Äquivalent:* Edit/Undo
CM_EDITFIND	TEditWindow::CMEditFind *Menü-Äquivalent:* Edit/Find
CM_EDITREPLACE	TEditWindow::CMEditReplace *Menü-Äquivalent:* Edit/Replace
CM_EDITFINDNEXT	TEditWindow::CMEditFindNext *Menü-Äquivalent:* Edit/Find/Next
CM_FILENEW	TFileWindow::CMFileNew *Menü-Äquivalent:* File/New
CM_FILEOPEN	TFileWindow::CMFileOpen *Menü-Äquivalent:* File/Open
CM_FILESAVE	TFileWindow::CMFileSave *Menü-Äquivalent:* File/Save
CM_FILESAVEAS	TFileWindow::CMFileSaveAs *Menü-Äquivalent:* File/Save As
CM_ARRANGEICONS	TMDIFrame::CMArrangeIcons *Menü-Äquivalent:* Window/Arrange Icons
CM_TILECHILDREN	TMDIFrame::CMTileChildren *Menü-Äquivalent:* Window/Tile
CM_CASCADECHILDREN	TMDIFrame::CMCascadeChildren *Menü-Äquivalent:* Window/Cascade
CM_CLOSECHILDREN	TMDIFrame::CMCloseChildren *Menü-Äquivalent:* Window/Close All
CM_CREATECHILD	TMDIFrame::CMCreateChild *Menü-Äquivalent:* Window/Create Child
CM_EXIT	TWindowsObject::CMExit *Menü-Äquivalent:* Window/Exit

__DELTA-Makro

objstrm.h

Beschreibung Makro zur Berechnung des Offset (in Byte) vom Beginn des Speicherbereichs eines stream-fähigen Objekts bis zur *TStreamable*-Komponente dieses Objekts. Das Argument ist der Name der stream-fähigen Klasse.

Querverweis *TStreamableClass*

dialogClass-Konstante

owldefs.h

Beschreibung Ein Klassenbezeichner zur Identifizierung von Instanzen der Klasse *TDialog*. Diese Konstante wird von der Elementfunktion *isA* zurückgeliefert, um für die Container-Klassen den Objekttyp zu bezeichnen.

Querverweis *TDialog::isA*

__DLL__-Makro

Beschreibung Dieses Makro wird automatisch gesetzt, wenn ein dynamisches Linkbibliotheksmodul mit den Optionen *-WD* oder *-WDE* compiliert wird. Andere Makros und bedingte Compilierungen legen mit Hilfe von *__DLL__* fest, ob eine dynamische Linkbibliothek aufgebaut werden soll.

EM_XXXX-Konstanten

owldefs.h

Beschreibung Folgende Fehlercodes sind definiert:

Tabelle 24.5 Konstanten für Fehlerbedingungen

24 **dialogClass-Konstante** 431

EM_INVALIDCHILD	-1	Eines oder mehrere der untergeordneten Fenster sind ungültig.
EM_INVALIDCLIENT	-2	Ein untergeordnetes MDI-Fenster konnte nicht erstellt werden.
EM_INVALIDMAINWINDOW	-3	Das Hauptfenster konnte nicht erstellt werden.
EM_INVALIDMODULE	-4	Das Fenster ist ungültig, da das Modulobjekt nicht gesetzt wurde.
EM_INVALIDWINDOW	-5	Das Fenster ist ungültig, da *Create* keinen Erfolg hatte.
EM_OUTOFMEMORY	-6	Bei einer Speicherreservierung wurde ein Teil des Sicherheitsbereichs belegt.

*EM_*-Konstanten dienen zur Kennzeichnung von Fehlerbedingungen, die von ObjectWindows entdeckt wurden.

_EXPORT-Makro

owldefs.h

Beschreibung Dieses Makro erweitert bedingt zu *_export*, wenn eine dynamische Linkbibliothek aufgebaut werden soll (*__DLL__* ist gesetzt). Andernfalls erweitert es zu *_CLASSTYPE*.

_FAR-Makro

_defs.h

Beschreibung

Dieses Makro steuert die Größe der Zeiger in Daten und Code dynamischer Linkbibliotheken. DLL-Programme benötigen Zeiger von Typ far, die andernfalls vom Typ near wären (in den Modellen *small* und *medium*).

_FAR wird zu far, wenn die Makros *__DLL__* und *_CLASSDLL* definiert sind. *_FAR* wird in Parameterlisten, Rückgabewerten, far/near-Aufrufkonventionen von Funktionen und Elementfunktionen und Klassendatenelementen, die Zeiger sind, verwendet. Die Typdefinitionen in der folgenden Tabelle sind Abkürzungen für häufig verwendete Typen, die eingesetzt werden, wenn eine Kompatibilität mit dynamischen Linkbibliotheken erforderlich ist.

Tabelle 24.6 DLL-kompatible typedefs

432 **Weitere Komponenten von ObjectW indows** 24

PCchar	`typedef const char _FAR * PCchar;` *Bedeutung:* Zeiger auf ein konstantes char	clstypes.h
Pchar	`typedef char _FAR * Pchar;` *Bedeutung:* Zeiger auf ein char	clstypes.h
PCvoid	`typedef const void _FAR * PCvoid;` *Bedeutung:* Zeiger auf eine Konstante mit unbestimmten Typ	clstypes.h
Pvoid	`typedef void _FAR * Pvoid;` *Bedeutung:* Zeiger auf einen unbestimmten Typ	clstypes.h
Rint	`typedef int _FAR & Rint;` *Bedeutung:* Referenz auf ein int	owldefs.h
RPvoid	`typedef void _FAR * _FAR & RPvoid;` *Bedeutung:* Referenz auf einen Zeiger mit unbestimmtem Typ	clstypes.h
RTMessage	`typedef TMessage _FAR & RTMessage;` *Bedeutung:* Referenz auf eine *TMessage*-Struktur	windobj.h

GetApplicationsObject()

Beschreibung

Diese Funktion kann aus einer Anwendung oder DLL heraus aufgerufen werden. Falls das aktive Fenster eine ObjectWindows-Anwendung ist, liefert *GetApplicationObject* einen Zeiger auf das *TApplication*-Objekt dieser Anwendung zurück, sonst *NULL*.

ID_XXXX-Konstanten

owldefs.h

Beschreibung

Die folgenden Konstanten definieren Werte oder Bereiche für Botschaften, die auf der ID eines untergeordneten Fensters basieren:

Tabelle 24.7 Konstanten für Botschaften untergeordneter Fenster

24 **ID_XXXX-Konstanten** 433

ID_COUNT	0x1000	Anzahl der Botschaften untergeordneter Fenster.
ID_FIRST	0x8000	Beginn der Botschaften untergeordneter Fenster.
ID_INTERNAL	0x8F00	Für internen Gebrauch reserviert.
ID_RESERVED		ID_INTERNAL - ID_FIRST;

Die folgende Tabelle enthält die voreingestellten Bezeichner für untergeordnete Fenster:

Tabelle 24.8 Basisikonstanten für ID_-Befehlsbotschaften

Konstante	Wert	Beschreibung
ID_FIRSTMDICHILD	ID_RESERVED+1	ID des untergeordneten MDI-Fensters.
ID_MDICLIENT	ID_RESERVED+2	ID des MDI-Client-Fensters.

ObjectWindows definiert zwei Kategorien von *ID_*-Konstanten. Eine Kategorie enthält die Konstanten, die die zulässigen Bereiche für die Bezeichner untergeordneter Fenster festlegen.

Die zweite Kategorie enthält die Bezeichner, die ObjectWindows standardmäßig für untergeordnete Fenster verwendet.

__link-Makro

objstrm.h

Beschreibung

Dieses Makro zwingt den Linker, die Funktionen einzubinden, die für das Lesen oder Schreiben einer gegebenen streamfähigen Klasse erforderlich sind. Das Argument ist der Name des statischen Registrierungsobjekts *TStreamableClass* (dieses heißt nach Konvention *RegClassName* für die Klasse *TClassName*).

moduleClass-Konstante

owldefs.h

Beschreibung

Ein Klassenbezeichner zur Identifizierung von Instanzen der Klasse *TModule*. Diese Konstante wird von der Elementfunkti-

on *isA* zurückgeliefert, um für die Container-Klassen den Objekttyp zu bezeichnen.

Querverweis *TModule::isA*

NF_XXXX-Konstanten

owldefs.h

Beschreibung Folgende Konstanten sind definiert:

Tabelle 24.9 Konstanten für Benachrichtigungsbotschaften

434 **Weitere Komponenten von ObjectW indows** 24

NF_COUNT	0x1000	Anzahl der Benachrichtigungsbotschaften.
NF_FIRST	0x9000	Beginn der Benachrichtigungsbotschaften.
NF_INTERNAL	0x9F00	Beginn der Benachrichtigungsbotschaften, die für den internen Gebrauch reserviert sind.

*NF_*-Konstanten definieren für Benachrichtigungsbotschaften einen Bereich mit Positionen zwischen 0x9000 und 0x9FFF, wobei 0x9F00-0x9FFF für den internen Gebrauch reserviert ist.

operator delete

Deklaration

```
void operator delete(Pvoid ptr);
```

ObjectWindows überschreibt den globalen Operator *delete*, um den Sicherheitsbereich nach jeder Speicherfreigabe zu überprüfen.

Wenn der Sicherheitsbereich erschöpft ist, versucht *delete* ihn neu anzulegen. Dadurch wird automatisch immer dann versucht den Sicherheitsbereich zu restaurieren, sobald dies erfolgversprechend erscheint.

Querverweis *SafetyPool::IsExhausted*

operator new

Deklaration

```
Pvoid operator new(size_t s);
```

ObjectWindows überschreibt den globalen Operator *new* zur Verwendung des Sicherheitsbereichs. Wenn eine Reservierung fehlschlägt, löscht *new* den bestehenden Sicherheitsbereich (falls vorhanden). Danach wird versucht, dem Objekt erneut Speicherplatz zu reservieren. Falls dies nicht möglich ist, wird 0 zurückgeliefert. Sobald der Sicherheitsbereich freigegeben ist, tritt eine »low memory«-Situation auf.

Weitere Aufrufe von *SafetyPool::IsExhausted* liefern 1 zurück.

Querverweis *SafetyPool::IsExhausted*

OWLGetVersion-Funktion

owl.h

Deklaration

```
WORD FAR OWLGetVersion();
```

Diese Funktion liefert die Versionsnummer der ObjectWindows-Bibliothek. Der Rückgabewert ist der Wert der Konstanten *OWLVersion*

OWLVersion-Konstante

owl.h

Beschreibung Mit Hilfe dieser in owl.h definierten Konstanten können Sie sicherstellen, daß die durch einen Aufruf der Funktion *OWLGetVersion* ermittelte Versionsnummer der erwarteten Nummer entspricht. Eine solche Überprüfung ist vor allem bei der Verwendung der ObjectWindows-DLL empfehlenswert.

P_id_type

objstrm.h

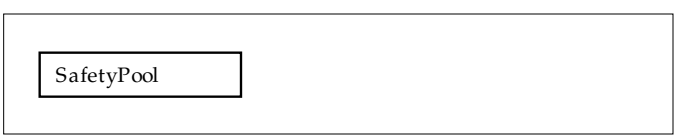
Definition

```
typedef unsigned P_id_type;
```

Die Stream-Verwaltung von *ObjectWindows* hält eine Datenbank mit den gelesenen und geschriebenen stream-fähigen Objekten. Dieser Typ dient zur Identifizierung solcher Objekte.

safetyPool-Klasse

safepool.h



Diese Klasse implementiert den Mechanismus des Sicherheitsbereichs, der von ObjectWindows verwendet wird, um Schnittstellenelemente in einer einzigen atomaren Operation zu erstellen.

Datenelemente

safetyPool

```
static Pvoid safetyPool;
```

Dieser Zeiger auf das Sicherheitsbereichobjekt wird zur Laufzeit auf einen Speicherblock gesetzt, der die Größe von *Size* hat. Anwendungen, die die Möglichkeit des Sicherheitsbereichs zum Abfangen von »Low Memory«-Bedingungen ausnützen,

können durch einen Aufruf von *IsExhausted* feststellen, ob der Sicherheitsbereich erschöpft ist.

Zur Änderung der Größe des Sicherheitsbereichs wird *Allocate* verwendet.

Querverweis: *SafetyPool::IsExhausted, SafetyPool::Size, SafetyPool::Allocate*

Size

```
static int Size;
```

Enthält die aktuelle Größe des Sicherheitsbereichs, der von der Klassenbibliothek zur Laufzeit mit dem Wert der Konstante *DEFAULT_SAFETY_POOL_SIZE* initialisiert wird.

Querverweis: *SafetyPool::Allocate, SafetyPool::SafetyPool*

Element-funktionen

Allocate

```
static int Allocate(size_t size =
DEFAULT_SAFETY_POOL_SIZE);
```

Löscht den vorhandenen Sicherheitsbereich und reserviert ihn mit der angebenen Größe.

Ist keine Größe angegeben, wird per Voreinstellung der Wert der Konstante *DEFAULT_SAFETY_POOL_SIZE* eingesetzt.

Die Funktion liefert bei Erfolg 1 zurück, andernfalls 0.

IsExhausted

```
static int IsExhausted();
```

Diese statische Elementfunktion liefert 1 zurück, wenn der Sicherheitsbereich erschöpft ist; andernfalls wird 0 zurückgegeben. Der Sicherheitsbereich gilt als erschöpft, wenn eine Speicherplatzreservierung die Freigabe des Sicherheitsbereichs zur Folge hat, wobei *safetyPool* auf 0 gesetzt wird.

Wenn diese Bedingung eintritt, bleibt die »Low Memory«-Situation weiterhin bestehen, bis ein erfolgreicher Aufruf von *Allocate* erfolgt ist.

Querverweis: *SafetyPool::Allocate, new*-Operator

scrollerClass-Konstante

owldefs.h

Beschreibung

Ein Klassenbezeichner zur Identifizierung von Instanzen der Klasse *TScroller*.

Diese Konstante wird von der Elementfunktion *isA* zurückgeliefert, um für die Container-Klassen den Objekttyp zu bezeichnen.

Querverweis

TScroller::isA

SD_XXXX-Konstanten

owlrc.h

Beschreibung Folgende Konstanten sind definiert:

Tabelle 24.10 Konstanten für Standarddialogfenster

24 SD_XXXX-Konstanten 437

SD_FILEOPEN	32512
SD_FILESAVE	32513
SD_INPUTDIALOG	32514
SD_SEARCH	32528
SD_REPLACE	32529

*SD_*-Konstanten sind die Bezeichner für ObjectWindows-Dialoge. Dialog-Ressourcen für *SD_SEARCH* und *SD_REPLACE* sind in der Datei STDWNDS.DLG enthalten.

StreamableInit-Typ

objstrm.h

Definition
```
enum StreamableInit{StreamableInit};
```

Zur Implementierung stream-fähiger Objekte gehört in jeder Klasse ein eindeutiger Konstruktor für die Initialisierung von Objekten, die aus Objekt-Streams eingelesen werden.

Die Parameterliste dieses Konstruktors besteht aus einem Argument des Typs *StreamableInit*.

TActionFunc-Typ

windobj.h

Definition
```
typedef void (_FAR *TActionFunc) (Pvoid Child,
                Pvoid ParamList);
```

Ein *TActionFunc*-Funktionszeiger wird an *TWindowsObject::ForEach*-Funktionen übergeben.

TActionMemFunc-Typ

windobj.h

Definition
```
typedef void(TWindowsObject::* _FAR TActionMemFunc)
              (Pvoid Child, Pvoid ParamList);
```

Ein *TActionMemFunc*-Funktionszeiger wird an *TWindowsObject::ForEach*-Funktionen übergeben.

TComboBoxData-Klasse

combobox.h

Beschreibung Ein *TComboBoxData*-Objekt ist ein Schnittstellenobjekt, das den Transferpuffer für ein *TComboBox* repräsentiert.

Datenelemente

Selection `Pchar Selection;`

Ein Zeiger auf den aktuell ausgewählten String, der in das oder aus dem Kombinationsfenster transferiert werden soll.

Strings `PArray Strings;`

String-Array für den Transfer in ein oder aus einem Kombinationsfenster.

Elementfunktionen

Konstruktor `TComboBoxData();`

Erstellt ein *TComboBox*-Objekt. Initialisiert *Strings* als leeres *Array* und *Selection* mit *NULL*.

Destruktor `~TComboBoxData();`

Löscht den für *Strings* und *Selection* reservierten Speicherplatz.

AddString `void AddString(Pchar AString, BOOL IsSelected = FALSE);`

Fügt den angegebenen String in das *Strings*-Array ein.
Wenn *IsSelected TRUE* ist, wird *Selection* gelöscht und anschließend auf eine Kopie von *AString* gesetzt.

TCondFunc-Typ

windobj.h

Definition
```
typedef BOOL (_FAR* TCondFunc) (Pvoid Child, Pvoid
ParamList);
```

Ein *TCondFunc*-Funktionszeiger wird an die *TWindowsObject::FirstThat*-Elementfunktion übergeben.

Querverweis *TWindowsObject::FirstThat*

TCondMemFunc-Typ

windobj.h

Definition
```
typedef BOOL (TWindowsObject::* _FAR TCondMemFunc)
             (Pvoid Child, Pvoid ParamList);
```

Ein *TCondMemFunc*-Funktionszeiger wird an die *TWindowsObject::FirstThat*-Elementfunktion übergeben.

Querverweis *TWindowsObject::FirstThat*

TDialogAttr-Typ

dialog.h

Definition

```
struct _CLASSTYPE TDialogAttr {
   LPSTR Name;
   DWORD Param;
};
```

In *TDialogAttr* werden die Erstellungsattribute für ein *TDialog*-Objekt gespeichert. Das Datenelement *Name* enthält den Bezeichner für die Dialog-Ressource. Das Datenelement *Param* speichert einen Parameter, welcher der Dialogprozedur zur Verfügung steht, wenn der Dialog erstellt wird. Windows akzeptiert *Param*. *Param* steht dann in den zu *WM_INITDIALOG* gehörigen Botschaftsantwortfunktionen zur Verfügung.

Querverweis *TDialog::Attr*

TF_XXXX-Konstanten

owldefs.h

Beschreibung Folgende Konstanten sind definiert:

Tabelle 24.11 Konstanten für die Transfer-Funktion

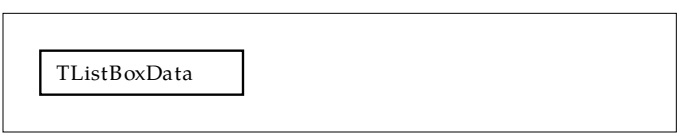
24 **TDialogAttr-Typ** 439

TF_GETDATA	0	Holt Daten aus der Klasse
TF_SETDATA	1	Sendet Daten an die Klasse
TF_SIZEDATA	2	Liefert den von der Klasse übertragenen Datenumfang zurück

*TF_*-Konstanten werden an die Elementfunktionen der Fensterklassen *Transfer* übergeben, um die auszuführende Operation anzugeben.

TListBoxData-Klasse

listbox.h



Ein *TListBoxData*-Objekt ist ein Schnittstellenobjekt, das einen Transferpuffer für ein *TListBox*-Objekt repräsentiert.

Datenelemente

SelCount

```
int SelCount;
```

Die Anzahl der ausgewählten Einträge.

SelStrings
```
PArray SelStrings;
```
Zeiger auf ein Array mit Strings, die bei der Übertragung von Daten in die Liste ausgewählt werden. Wenn die Daten aus der Liste geholt werden, liefert *SelStrings* die aktuelle(n) Auswahl(en).

Strings
```
PArray Strings;
```
Zeiger auf ein Array mit Strings, die in ein *TListBox*-Objekt übertragen werden sollen.

Elementfunktionen

Konstruktor
```
TListBoxData();
```
Erstellt *Strings* und *SelStrings* und initialisiert *SelCount* mit 0.

Destruktor
```
~TListBoxData();
```
Löscht den für *Strings* und *SelStrings* reservierten Speicherplatz.

AddString
```
void AddString(PChar AString, BOOL IsSelected = FALSE);
```
Fügt den angegebenen String in *Strings* ein. Wenn *IsSelected* *TRUE* ist, wird der String in *SelStrings* eingefügt und *SelCount* hochgezählt.

GetSelString
```
void GetSelString(LPSTR Buffer, int BufferSize,
                  int Index = 0);
```
Kopiert den String an der Position *Index* von *SelStrings* nach *Buffer*. *BufferSize* beinhaltet die abschließende Null.

GetSelStringLength
```
int GetSelStringLength(int Index = 0);
```
Liefert die Länge des Strings (ohne die abschließende Null) an der in *Index* angegebenen Position in *SelStrings* zurück.

ResetSelections
```
void ResetSelections();
```
Entfernt alle Strings aus *SelStrings* und setzt *SelCount* auf 0.

SelectString
```
void SelectString(LPSTR AString);
```
Fügt *AString* in *SelStrings* ein und zählt *SelCount* hoch.

TMessage-Typ

windobj.h

Definition
```
struct TMessage  {
   HWND Receiver;
   WORD Message;
   union {
      WORD WParam;
      struct tagWP  {
         BYTE Lo;
```

```
      BYTE Hi;
    } WP;
  };
  union {
    DWORD LParam;
    struct tagLP {
      WORD Lo;
      WORD Hi;
    } LP;
  };
  long Result;
};
```

Eine *TMessage*-Struktur enthält Informationen über eine eingehende Windows-Botschaft. Die Strukturen werden von ObjectWindows ausgefüllt und an Botschaftsanwortfunktionen übergeben. *Receiver* enthält das Handle des Fensters, für das die Meldung ursprünglich gedacht ist. *Message* nimmt den Botschaftsbezeichner auf. Die Datenelemente *WParam* und *LParam* enthalten botschaftsspezifische Windows-Informationen. Die *LP*- und *WP*-Strukturen verschaffen einen einfachen Zugriff auf die höher- und niederwertigen Bereiche von *WParam* und *LParam*. In *Result* steht der Wert, der an Windows zurückgeliefert wird.

TScrollBarData-Typ

scrollba.h

Definition

```
struct TScrollBarData {
  int LowValue;
  int HighValue;
  int Position;
};
```

Eine *TScrollBarData*-Struktur wird zum Speichern der *TScrollBar*-Daten verwendet. Mit *LowValue* und *HighValue* wird der Bildlaufbereich gesetzt und geholt. *Position* dient zur Ermittlung und Festlegung der Position des Positionszeiger.

Querverweis *TScrollBar::Transfer*

TSearchStruct-Typ

editwnd.h

Definition

```
struct _CLASSTYPE TSearchStruct {
  char SearchText[81];
  BOOL CaseSensitive;
  char ReplaceText[81];
  BOOL ReplaceAll;
  BOOL PromptOnReplace;
};
```

Ein *TSearchStruct* enthält den Suchen-/Ersetzen-Text und die damit verbundenen Optionen, die von einem *TSearchDialog* übergeben werden.

SearchText (zu suchender Text) und *ReplaceText* (ersetzt *SearchText*) enthalten die vom Anwender eingegebenen Strings. Die weiteren Parameter sind vom Anwender ausgewählte Optionen für die Berücksichtigung der Groß-/Kleinschreibung (*CaseSensitive*) und der globalen Ersetzung (*ReplaceAll*). *PromptOnReplace* zeigt an, ob der Anwender eine Ersetzung bestätigen soll.

Querverweis *TSearchDialog::TSearchDialog, TEditWindow::CMEditFind, TEditWindow::CMEditReplace*

TWindowAttr-Typ

window.h

Definition

```
struct _CLASSTYPE TWindowAttr  {
   DWORD Style;
   DWORD ExStyle;
   int X, Y, W, H;
   LPSTR Menu;
   int Id;
   LPSTR Param;
};
```

TWindowAttr wird für die Speicherung der *TWindow*-Erstellungsattribute verwendet.

Style und *ExStyle* enthalten das Fenster und die erweiterten Stile. *X* und *Y* beinhalten die Bildschirmkoordinaten für die obere linke Ecke des Fensters. In *W* und *H* sind Breite und Höhe des Fensters gespeichert. *Param* enthält einen Wert, der bei der Erstellung des Fensters an Windows übergeben wird. Er steht dann den mit *WM_CREATE* verknüpften Botschaftsantwortfunktionen zur Verfügung.

Param wird von *TMDIClient* verwendet und kann die Konvertierung von Nicht-ObjectWindows-Code unterstützen.

Menu beinhaltet den Ressourcen-Bezeichner des Fenstermenüs (falls vorhanden). *Id* enthält den Bezeichner des Fensters; für ein Dialog-Steuerelement enthält *Id* den Ressourcen-Bezeichner.

Querverweis *TWindow::Attr*

WB_XXXX-Konstanten

owldefs.h

Beschreibung Es sind die in der folgenden Tabelle aufgeführten Werte definiert:

24 **windowClass-Konstante** *443*

Konstanten	Wert	Beschreibung der entsprechenden Attribute (falls gesetzt)
WB_ALIAS	0x01	Ist mit einem Fenster des Aufrufers einer DLL verknüpft.
WB_AUTOCREATE	0x02	Zusammen mit dem übergeordneten Fenster erstellt.
WB_FROMRESOURCE	0x04	Über eine Ressourcen-Definition erstellt.
WB_KBHANDLER	0x08	Tastaturschnittstelle für Steuerelemente.
WB_MDICHILD	0x10	Ist ein untergeordnetes MDI-Fenster.
WB_MDIFRAME	0x20	Ist ein MDI-Hauptfenster (Rahmenfenster).
WB_PREDEFINEDCLASS	0x40	Hat vordefinierte Fensterklasse.
WB_TRANSFER	0x80	Nimmt am *Transfer* teil.

*WB_*-Konstanten definieren Bit-Masken für die intern verwendeten Flag-Attribute des *TWindowsObject*-Objekts.

Für jedes dieser Attribute ist eine *WB_*-Maske definiert.

Querverweis *TWindowsObject::SetFlags*, *TWindowsObject::EnableAutoCreate*, *TWindowsObject::EnableKBHandler*, *TWindowsObject::EnableTransfer*

windowClass-Konstante

owldefs.h

Beschreibung Ein Klassenbezeichner, der zur Identifizierung der Instanzen der Klasse *TWindow* dient. Diese Konstante wird von der Elementfunktion *isA* zurückgeliefert, um der Container-Klasse den Typ des Objekts mitzuteilen.

Querverweis: *TWindow::isA*

WM_XXXX-Konstanten

owldefs.h

Beschreibung Folgende Konstanten sind definiert:

Tabelle 24.13 Konstanten von Windows-Botschaften

Konstante	Wert	Beschreibung
WM_COUNT	0x8000	Anzahl der reservierten Botschaften
WM_FIRST	0x0000	Anfang des reservierten Bereichs
WM_INTERNAL	0x7F00	Beginn des reservierten Object-Windows-Bereichs
WM_RESERVED		*WM_INTERNAL-WM_FIRST*

ObjectWindows definiert Konstanten, die mit den Windows-Botschaften *WM_XXXX* verwandt sind. Der Bereich 0x7F00 bis 0x7FFF ist für den internen Gebrauch von ObjectWindows reserviert (vgl. Kapitel 2 »Windows-Botschaften« im *Windows-Referenzhandbuch*).

Index

&-Zeichen 167
< Operator 318
> Operator 318

A
Abbildungsmodi 263
abstrakte Klasse
  Object 316
ActivationResponse
  TWindow-Elementfunktion 390
  TWindowsObject-Elementfunktion 398
ActiveChild
  TMDIFrame-Datenelement 365
AddString
  TComboBoxData-Elementfunktion 438
  TListBox-Elementfunktion 159, 359
  TListBoxData-Elementfunktion 440
AfterDispatchHandler
  TWindowsObject-Elementfunktion 398
Aktionsschalter 323ff.
  Beispiel 178
  Botschaften und 174
  erstellen 174
  Stile 174, 323ff.
  voreingestellte 174
Allocate
  SafetyPool-Elementfunktion 436
AModule
  explizit übergeben 397
andauernder Datentransfer 257
Anforderungen
  Hardware 3
  Software 3
AnimatePaletteEntries-Funktion 283
Anschrift von Borland 7
anwenderdefinierte Steuerelement-
  objekte 191
Anwendungen 11
  »Hello, World« 20
  Anforderungen 12
  Aufbau 24
  beenden 37
  Botschaften 321
  entwickeln 22
  Fehler 371
  Hauptfenster 104
  initialisieren 104ff., 320
  Konstruktor 31, 103
  Objekte 30f., 86, 103ff., 308
  schließen 109, 320, 349
  sichere 306
  Start 21
  Status 31
  Struktur 19
  Systemmenüsymbol 106
  Voraussetzungen 30
  Vorteile von Windows 12
Anwendungsobjekt 30f., 86, 103ff. 308
API (Windows) 91
applicationClass-Konstante 321, 427
Arc-Funktion 277
ArrangeIcons
  TMDIClient-Elementfunktion 364
  TMDIFrame-Elementfunktion 367
Array-Klasse 212
AssignMenu
  TWindow-Elementfunktion 59, 391
attach
  fpbase-Elementfunktion 412
Attr
  TDialog-Datenelement 334
  TWindow-Datenelement 122, 389
Attribute
  Client-Fenster 363
  Dialoge 334
  Erstellungs- 126
  Registrierung 126ff., 295
  siehe auch Registrierungsattribute

Aufzählungsfunktionen 94
Auswahlfelder 175, 325
  Beispiel 178
  Benachrichtigungsbotschaften 326
  erzeugen 175
  Status 325
  Stile 176, 325
  Transferpuffer 186
  siehe auch Gruppen
  siehe auch Wahlfelder
automatischer Bildlauf 133, 379
AutoMode
  TScroller-Datenelement 133, 379
AutoOrg
  TScroller-Datenelement 380
AutoScroll
  TScroller-Elementfunktion 382

B
bad
  pstream-Elementfunktion 422
Befehle
  antworten auf 60
  Tastenkürzel und 294
BeforeDispatchHandler
  TWindowsObject-Elementfunktion 398
BeginView
  TScroller-Elementfunktion 382
Benachrichtigungs-Code 147
Benachrichtigungsbotschaften
  Bildlaufleisten und 183
Bereich
  Bildlaufleisten 179, 181
BF_CHECKED-Konstante 176, 326
BF_GRAYED-Konstante 176, 326
BF_UNCHECKED-Konstante 176, 326
BF_XXXX-Schalter-Flag-
  Konstanten 427
Bibliotheken
  C++ 25
  Container-Klassen 25
  Import 26
  ObjectWindows 25
  Speichermodelle und 25
Bildlauf 131ff., 379
  automatischer 133
  Beispiel 131
  Bereich 134, 441
  Bereich ändern 134
  Bildlaufleisten und 131
  in Fenstern 129
  Paint und 131
  Position ändern 134
  Schrittweite ändern 134
  Seitengröße ändern 135
  Zeichenelementfunktionen und 135
Bildlaufleisten 131 ff., 179ff., 375ff.
  ändern 181
  automatischer Bildlauf und 133
  Beispiel 184
  Bildlauf in Fenstern und 131
  Botschaften und 182
  erzeugen 180
  Informationen über 181
  Objekte 90
  Positionsverwaltung 375ff.
  Seitengröße 180
  Stile 179ff.
  Transferpuffer 186
  Zeilengröße 180
Bildschirm
  löschen 44, 61
Bildschirmkontext 41, 261ff.
  Bitmaps und 262
  Einheiten 263
  freigeben 262
  Funktionen 42
  Handles 41
  Hintergrundfarbe 263
  Koordinaten 263
  verwalten 262
  Zeichenwerkzeuge und 263ff.
  Zeichnen und 52
Bitmaps
  als Menüpunkte 298, 300
  als Pinsel 298
  Bildschirmkontext und 262
  geräteunabhängige 262
  Kompatibilität 262
  laden 297
  löschen 298
  vordefinierte 297
BMSetStyle
  TButton-Elementfunktion 324
BN_CLICKED-Benachrichtigung 156
BNClicked
  TCheckBox-Elementfunktion 326
  TRadioButton-Elementfunktion 374
Borland
  technische Beratung 7
Botschaften 14, 95
  Aktionsschalter 174
  anwenderdefinierte 100
  auf Befehlen basierend 400

auf IDs untergeord. Fenster
basierend 100, 116ff., 147, 174, 399
beantworten 35, 115
bearbeiten 108, 115
Befehls- 60, 100
befehlsabhängige 116
Benachrichtigungs- 100, 147, 156
Bereiche 100
Bildlaufleisten und 182
Botschaftsantwortfunktionen 14, 35, 115ff., 193
Dialogfenster und 321
Ereignisse und 95, 115
Fehlerkonstanten 431
Gruppen und 177ff., 356
Kombinationsfenster und 329
Listen und 349, 360
Listen-Benachrichtigungsbotschaften 161
Mausereignisse 45f.
Menüs und 60
Multidokumentschnittstelle 200
nachbearbeiten 398
Parameter 96, 116
senden 99
Standardantworten 99, 118
Standardbearbeitung 396
Steuerelemente und 80, 99, 145ff., 156
Struktur 43
Tastenkürzel und 108
TMessage-Typ 441
Typen 97
Übermittlung zwischen Anwendungen 249
Windows 95
Botschaftsantwortfunktionen 14, 35, 115ff.
Standardantwort ändern 193
Standardantwort überschreiben 193
Botschaftsschleife 108, 321
Windows 320
bp
pstream-Datenelement 422
BS_AUTOCHECKBOX-Stil 176
BS_AUTORADIOBUTTON-Stil 176
BS_DEFPUSHBUTTON-Stil 174, 323f.
BS_DIBPATTERN-Stil 267
BS_GROUPBOX-Stil 355
BS_HATCHED-Stil 267
BS_HOLLOW-Stil 267
BS_OWNERDRAW-Stil 193
BS_PATTERN-Stil 267
BS_PUSHBUTTON-Stil 174
Buffer
TInputDialog-Datenelement 356
BufferSize
TInputDialog-Datenelement 357
build
TButton-Elementfunktion 324
TCheckBox-Elementfunktion 326
TComboBox-Elementfunktion 329
TDialog-Elementfunktion 335
TEdit-Elementfunktion 341
TEditWindow-Elementfunktion 346
TFileDialog-Elementfunktion 349
TFileWindow-Elementfunktion 351
TGroupBox-Elementfunktion 356
TInputDialog-Elementfunktion 357
TListBox-Elementfunktion 359
TMDIClient-Elementfunktion 364
TMDIFrame-Elementfunktion 367
TRadioButton-Elementfunktion 374
TScrollBar-Elementfunktion 376
TScroller-Elementfunktion 382
TStatic-Elementfunktion 387
TWindow-Elementfunktion 391
TWindowsObject-Elementfunktion 398
build-Elementfunktion
für stream-fähige Klassen 208
Build-Konstruktor 208
BUILDER-Typ 428
Bytes
Größe der Transferdaten 327, 408
reservieren 317

C
Callback-Funktionen 93
_export und 94
Cancel
TDialog-Elementfunktion 144, 335
CanClear
TFileWindow-Elementfunktion 351
CanClose
TApplication-Elementfunktion 37, 109, 319
TFileDialog-Elementfunktion 349
TFileWindow-Elementfunktion 351
TWindowsObject-Elementfunktion 37, 69, 398
untergeordnete Fenster und 37
CanUndo
TEdit-Elementfunktion 170, 341
Caret
Position 344

CascadeChildren
TMDIClient-Elementfunktion 364
TMDIFrame-Elementfunktion 367
CBS_AUTOHSCROLL-Stil 165, 329
CBS_DROPDOWN-Stil 164, 328f.
CBS_DROPDOWNLIST-Stil 164, 328f.
CBS_OWNERDRAW-Stil 193
CBS_OWNERDRAWFIXED-Stil 329
CBS_OWNERDRAWVARIABLE-Stil 329
CBS_SIMPLE-Stil 164, 328f.
CBS_SORT-Stil 165, 329
CF_BITMAP-Konstante 245
CF_METAFILEPICT-Konstante 245
CF_SYLK-Konstante 245
CF_TEXT-Konstante 245f., 255
CF_TIFF-Konstante 245
Check
TCheckBox-Elementfunktion 176, 326
ChildMenuPos
TMDIFrame-Datenelement 199, 365
ChildWithId
TWindow-Elementfunktion 155
TWindowsObject-Elementfunktion 399
Chord-Funktion 279
_CLASSDEF(Klasssenname)-Makro 428
_CLASSDLL-Makro 428
ClassHashValue
TScroller-Datenelement 380
_CLASSTYPE-Makro 428
Clear
pstream-Elementfunktion 422
TComboBox-Elementfunktion 329f.
TEdit-Elementfunktion 173
TStatic-Elementfunktion 167, 387
ClearList
TListBox-Elementfunktion 160, 359
ClearModify
TEdit-Elementfunktion 172, 341
Client-Bereich 42
Bildlauf im 129, 131
Client-Fenster 250, 263ff.
Symbole 364
ClientAttr
TMDIClient-Datenelement 363
ClientWnd
TMDIFrame-Datenelement 365
Clipping-Regionen 263
close
fpbase-Elementfunktion 412
CloseChildren
TMDIFrame-Elementfunktion 367
CloseClipboard-Funktion 246f.
CloseWindow 144
TDialog-Elementfunktion 336
TWindowsObject-Elementfunktion 399
CM_ARRANGEICONS-Botschaft 367
CM_CASCADECHILDREN- Botschaft 367
CM_CLOSECHILDREN-Botschaft 367
CM_CREATECHILD-Botschaft 368
CM_EDITCLEAR-Konstante 341
CM_EDITCOPY-Konstante 342
CM_EDITCUT-Konstante 342
CM_EDITDELETE-Konstante 342
CM_EDITPASTE-Konstante 342
CM_EDITUNDO-Konstante 342
CM_EXIT-Botschaft 399
CM_FILENEW-Botschaft 352
CM_FILEOPEN-Botschaft 352
CM_FILESAVE-Botschaft 352
CM_FILESAVEAS-Botschaft 352
CM_FIRST-Konstante 117, 294
CM_TILECHILDREN-Botschaft 368
CM_XXXX-Konstanten 429
CMArrangeIcons
TMDIFrame-Elementfunktion 367
CMCascadeChildren
TMDIFrame-Elementfunktion 367
CMCloseChildren
TMDIFrame-Elementfunktion 367
CMCreateChild
TMDIFrame-Elementfunktion 368
CMEditClear
TEdit-Elementfunktion 341
CMEditCopy
TEdit-Elementfunktion 170, 342
CMEditCut
TEdit-Elementfunktion 342
CMEditDelete
TEdit-Elementfunktion 342
CMEditFind
TEditWindow-Elementfunktion 346
CMEditFindNext
TEditWindow-Elementfunktion 347
CMEditPaste
TEdit-Elementfunktion 342
CMEditReplace
TEditWindow-Elementfunktion 347
CMEditUndo
TEdit-Elementfunktion 170, 342
CMExit
TWindowsObject-Elementfunktion 399
CMFileNew
TFileWindow-Elementfunktion
139, 352

CMFileOpen
TFileWindow-Element-funktion 139, 352
CMFileSave
TFileWindow-Element-funktion 139, 352
CMFileSaveAs
TFileWindow-Element-funktion 139, 352
CMFileWindow
TFileWindow-Elementfunktion 139
CMTileChildren
TMDIFrame-Elementfunktion 201, 368
Codes
Fehler- 371, 431
Common User Access (CUA) 301
Container-Klassenbibliothek 24
Array 53
Object 53
Object-Klasse 86
Copy
TEdit-Elementfunktion 170, 342
Create
Aufruf durch andere Element-funktionen 125
MakeWindow und 125
MakeWindow vs. 65, 307
TDialog-Elementfunktion 336
TWindow-Elementfunktion 391
TWindowsObject-Elementfunktion 399
Create-Elementfunktion 112
CreateChild
TMDIFrame-Elementfunktion 199, 368
CreateChildren
TWindowsObject-Elementfunktion 399
CreatePalette-Elementfunktion 282
CreatePen-Funktion 51, 265
CreatePenIndirect-Funktion 265
CS_-Konstanten 128
CS_HREDRAW-Stil 392
CS_VREDRAW-Stil 392
Cut
TEdit-Elementfunktion 342

D
Datei-Dialoge 151
Erstellung 67, 151
Dateien
.RC 289
.RES 288
Dialoge und 348
editieren 352
Extension 348
Header- 291
Listen und 350
öffnen 67, 151, 348, 352
Ressourcen-Script 288
sichern 67, 152, 348
Streams und 412ff.
Textdateien editieren 136
Dateifenster 136, 139, 350
Dateinamen
Editierfelder und 350
Datenelemente 315
Objekt hinzufügen 67
untergeordnete Fenster und 78
Datenkonvertierung
verzögerte 249
DDE;
siehe Dynamischer Datenaustausch
dde.h-Datei 250
DefaultProc
TWindowsObject-Datenelement 396
DefChildProc
TWindowsObject-Elementfunktion 399
DefCommandProc
TWindowsObject-Elementfunktion 400
DefNotificationProc
TWindowsObject-Elementfunktion 400
DefWndProc
TWindowsObject-Elementfunktion 99, 116, 400
DeleteLine
TEdit-Elementfunktion 173, 342
DeleteObject-Funktion 298
DeleteSelection
TEdit-Elementfunktion 173, 343
DeleteString
TListBox-Elementfunktion 160, 359
DeleteSubText
TEdit-Elementfunktion 173, 343
__DELTA-Makro 430
TStreamableClass und 426
DeltaPos
TScrollBar-Elementfunktion 181, 376
Destroy
TDialog-Elementfunktion 336f.
TWindowsObject-Elementfunktion 400
dialogClass-Konstante 431
Dialoge 63
Attribute 334
ausführen 337
modale und nicht-modale 334
siehe auch Dialogfenster

Dialogfenster 63f., 141ff.
als untergeordnete Fenster 142
anzeigen 67
ausführen 142
Datei- 66
erzeugen 67, 142
Fenster vs. 142
laden 295
modale und nicht-modale 108, 143ff.
Objekte 88, 142
Pop-up-Fenster und 66
Ressourcen und 66, 141, 295
Rückgabewerte 66
schließen 144, 339
SD_XXXX-Konstanten und 437
Standard- 305
Steuerelemente und 145
Stringtabellen und 296
Transfermechanismus 186
vordefiniertes 66
zum Ersetzen 385
zum Suchen 385
DisableAutoCreate
TWindowsObject-Element-funktion 114, 401
DisableTransfer
TControl-Elementfunktion 187
TWindowsObject-Elementfunktion 401
DispatchAMessage
TWindowsObject-Elementfunktion 401
DispatchScroll
TWindowsObject-Elementfunktion 401
__DLL__-Makro 431
DLLs 233ff., 241ff.
AModule und 397
Austrittspunkt 234
Haupteinsprung 234
Initialisierungsfunktionen 235
kompatible Typdefinitionen 432
LibMain-Funktion 234
Module 370ff.
Objektzeiger 372
WEP-Funktion 234
DOS 19
DoSearch
TEditWindow-Elementfunktion 347
DrawItem
TWindowsObject-Elementfunktion 401
Dynamische Linkbibliotheken
siehe DLLs
Dynamischer Datenaustausch (DDE) 245ff.
andauernder Datentransfer 257
Botschaften 250
Elementfunktionen 254
globale Atome und 250
Hot Links 254
Konversation 251
Makrobefehle 259
System-Thema 260

E
Editierfelder 167, 340ff.
ändern 172
Beispiel 174
Bildlauf 173
Editierfenster und 136
Editor 346
erzeugen 168
Informationen über 170
löschen 173
mehrzeilige 168, 343, 346
Text bearbeiten 341ff.
Text einfügen 173
Text löschen 173
Text zurückliefern 171
Textauswahl 173
Transferpuffer 185
Zeilen ermitteln 171
Zwischenablage und 170
Editierfenster 136
Dateifenster und 139
Editor
TEditWindow-Datenelement 346
TFileWindow-Datenelement 140
Eingabe-Dialoge 149, 356ff.
Eingabefelder 49ff.
Elementfunktionen 14, 313, 315
Botschaftsantwort 35, 115
Ellipse-Funktion 279
EM_INVALIDCHILD-Konstante 309, 396, 408
EM_INVALIDCLIENT-Konstante 309, 396
EM_INVALIDMAINWINDOW-Konstante 309, 320, 396
EM_INVALIDMODULE-Konstante 371, 396f.
EM_INVALIDWINDOW-Konstante 309, 336
EM_OUTOFMEMORY-Konstante 309
EM_XXXX-Fehlerbedingungs-konstanten 431
EmptyClipboard-Funktion 246

EnableAutoCreate
TWindowsObject-Element-funktion 114, 401
EnableKBHandler 158
TWindowsObject-Elementfunktion 402
EnableTransfer
TControl-Elementfunktion 187
TWindowsObject-Elementfunktion 402
EndView
TScroller-Elementfunktion 382
eof
pstream-Elementfunktion 423
ereignisgesteuerte Programmierung 115
Ereignisse 80
Botschaften und 115
Steuerelemente und 80
Error
pstream-Elementfunktion 423
TModule-Elementfunktion 371
Erstellungsattribute
Fenster 389
Erstellungsstil
ändern 191
ES_AUTOHSCROLL-Stil 168
ES_AUTOVSCROLL-Stil 169
ES_LEFT-Stil 168
ES_MULTILINE-Stil 169
ES_UPPERCASE-Stil 174
ExecDialog 51
Error und 308
Execute vs. 68
modaler Dialog und 143
Rückgabewerte 308
TApplication-Elementfunktion 68, 143
TModule-Elementfunktion 371
Execute
ExecDialog vs. 68
TDialog-Elementfunktion 337
_export
Callback-Funktionen 94
_EXPORT-Makro 431
Extension
TFileDialog-Datenelement 348

F
fail
pstream-Elementfunktion 423
_FAR-Makro 432 Farben 262, 266, 281ff.
Abwandlung 262
Bildschirmkontext und 261, 263
Hintergrund 126ff., 261, 263, 392
Intensität 265
setzen 265
Stifte 265
verwenden 306
Fehler
Behandlung 308
Fenster
anzeigen 73, 112
Attribute erstellen 126
aufbauen 393
bearbeiten 388ff.
benennen 127
Bildlauf in 129
Bildschirmkontext und 271
Client- 363
deaktivieren 143
editieren 345
erstellen 388ff.
Grafikanzeige in 270
Grafikfunktionen aufrufen 271
Hintergrundfarbe 129
Konstruktor 59
Koordinaten 42
Menüattribute 59
nebengeordnete 405
Neuzeichnen 270, 272
Objekte 33, 88
Pop-up-Fenster 73
Registrierung 126, 392
schließen 339
Sicherheitsbereich anlegen 306
Steuerelemente in 76
Stile 123
Stilkonstanten 326
Titel 123
Transfermechanismus 186
verändern 305
zeichnen in 271
Fensterelemente
erstellen 125
Fensterklassen
nicht-modale Dialoge 145
Fensterklassenregistrierung 407
Fensterobjekte 121
Attribute 122
initialisieren 122
Konstruktor 123
FileName
TFileWindow-Datenelement 351
FilePath
TFileDialog-Datenelement 348
FileSpec

TFileDialog-Datenelement 348
find
ipstream-Elementfunktion 416
opstream-Elementfunktion 420
FindExactString
TListBox-Elementfunktion 160, 359
FindString
TListBox-Elementfunktion 160, 360
FirstThat
Object-Elementfunktion 316
TWindowsObject-Elementfunktion 114, 402
Flags
globaler Speicher 227
lokaler Speicher 223
flush
opstream-Elementfunktion 420
FocusChildHandle
TWindow-Datenelement 389
Fokus
übergeben 333, 347
weitergeben 319, 349
ForEach
Object-Elementfunktion 317
TWindowsObject-Elementfunktion 114, 403
Formate
konvertieren 249
fortbestehende Objekte 203
fpbase-Klasse 412
freadBytes
ipstream-Elementfunktion 416
freadString
ipstream-Elementfunktion 416
Funktionen
API 93
Callback- 93
Windows- 91
fwriteBytes
opstream-Elementfunktion 420
fwriteString
opstream-Elementfunktion 420

G
GDI
CreatePen-Funktion 51
InvalidateRect-Funktion 44
LineTo-Funktion 47
MoveTo-Funktion 47
Stifte 50
Zeichenwerkzeuge 48
GDI.EXE 19
gemeinsam genutzte Klassen 237ff.
geometrische Formen 278
Gerätetreiber (Bildschirmkontext) 261
GetApplication
TWindowsObject-Elementfunktion 403
GetCheck
TCheckBox-Elementfunktion 176, 326
GetChildPtr
TWindowsObject-Elementfunktion 403
GetChildren
TWindowsObject-Elementfunktion 403
GetClassName 145
TButton-Elementfunktion 324
TComboBox-Elementfunktion 329
TDialog-Elementfunktion 337
TEdit-Elementfunktion 343
TGroupBox-Elementfunktion 356
TListBox-Elementfunktion 360
TMDIClient-Elementfunktion 364
TMDIFrame-Elementfunktion 368
TScrollBar-Elementfunktion 376
TStatic-Elementfunktion 387
TWindow-Elementfunktion 392
TWindowsObject-Elementfunktion 404
überschreiben 127
GetClient
TMDIFrame-Elementfunktion 368
TWindowsObject-Elementfunktion 404
GetClientHandle
TModule-Elementfunktion 371
GetClipboardData-Funktion 247
GetCount
TListBox-Elementfunktion 160, 360
GetEditSel
TComboBox-Elementfunktion 329
GetFirstChild
TWindowsObject-Elementfunktion 404
GetID
TControl-Elementfunktion 332
TWindowsObject-Elementfunktion 404
GetInstance
TWindowsObject-Elementfunktion 404
GetItemHandle
TDialog-Elementfunktion 337
GetLastChild
TWindowsObject-Elementfunktion 404
GetLine
TEdit-Elementfunktion 171, 343
GetLineFromPos
TEdit-Elementfunktion 343
GetLineIndex
TEdit-Elementfunktion 343

GetLineLength
TEdit-Elementfunktion 171, 344
GetModule
TWindowsObject-Elementfunktion 404
GetMsgID
TComboBox-Elementfunktion 329
TListBox-Elementfunktion 360
GetNumLines
TEdit-Elementfunktion 171, 344
GetParentObject
TModule-Elementfunktion 371
GetPosition
TScrollBar-Elementfunktion 181, 376
GetRange
TScrollBar-Elementfunktion 181, 377
GetSelCount
TListBox-Elementfunktion 360
GetSelection
TEdit-Elementfunktion 172, 344
GetSelIndex
TListBox-Elementfunktion 161f., 360
GetSelIndexes
TListBox-Elementfunktion 360
GetSelString
TListBox-Elementfunktion 162, 360
TListBoxData-Elementfunktion 440
GetSelStringLength
TListBoxData-Elementfunktion 440
GetSelStrings
TListBox-Elementfunktion 361
GetSiblingPtr
TWindowsObject-Elementfunktion 405
GetStockObject-Funktion 264
GetString
TListBox-Elementfunktion 161, 361
GetStringLen
TListBox-Elementfunktion 161, 361
GetSubText
TEdit-Elementfunktion 172, 344
GetText
TComboBox-Elementfunktion 330
TEdit-Elementfunktion 171
TStatic-Elementfunktion 167, 387
GetTextLen
TComboBox-Elementfunktion 330
TStatic-Elementfunktion 388
GetWindowClass 145, 295
Mauszeiger und 295
Symbole und 295
TDialog-Elementfunktion 337
TMDIFrame-Elementfunktion 368
TWindow-Elementfunktion 392
TWindowsObject-Element-funktion 128, 405
überschreiben 127
GlobalAlloc-Funktion 226
Rückgabewerte 227
GlobalCompact-Funktion 230
GlobalDiscard-Funktion 228
globale Atome (DDE) 250
globaler Heap
siehe globaler Speicher
globaler Speicher 221, 226ff.
64-KByte-Grenze 229
abfragen 229
Flags 226
freigeben 228
Größe 227, 229
nicht verschiebbarer 228
reservieren 226
Sperre aufheben 227f.
sperren 228
ungültiges Handle 227
verwerfen 230
Zeiger auf 227
zusammenfassen 230
GlobalFix-Funktion 228
GlobalFlags-Funktion 229
GlobalFree-Funktion 228
GlobalLock-Funktion 227
GlobalLRUNewest-Funktion 230
GlobalLRUOldest-Funktion 230
GlobalReAlloc-Funktion 228f.
GlobalSize-Funktion 229
GlobalUnix-Funktion 228
GlobalUnlock-Funktion 227
GlobalUnwire-Funktion 228
GlobalWire-Funktion 228
GMEM_DDESHARE-Flag 229, 250
GMEM_DDESHARE-Konstante 255
GMEM_DISCARDABLE-Flag 227, 229
GMEM_FIXED-Flag 227
GMEM_MOVEABLE-Flag 227, 229
GMEM_NOCOMPACT-Flag 227, 229
GMEM_NODISCARD-Flag 227, 229
GMEM_NOT_BANKED-Flag 229
GMEM_ZEROINIT-Flag 228
good
pstream-Elementfunktion 423
Grafische Geräte-schnittstelle (GDI) 261
Zeichenfunktionen 275
Group
TCheckBox-Datenelement 325

Gruppen 354ff.
  Auswahlfelder und 325
  Botschaften und 178, 356
  erstellen 177f.

H
HAccTable
  TApplication-Datenelement 294, 319
HandleDList
  TFileDialog-Elementfunktion 349
HandleFList
  TFileDialog-Elementfunktion 349
HandleFName
  TFileDialog-Elementfunktion 350
Handles
  Bildschirmkontext und 41
  Fenster- 34, 112, 371
  Instanzen 319
  Speicherverwaltung und 221
  Tastenkürzeltabelle 319
  Zeichenwerkzeuge 48
Hardware 3
HasHScrollBar
  TScroller-Datenelement 380
hashValue
  Object-Elementfunktion 317
  TModule-Elementfunktion 372
  TScroller-Elementfunktion 382
  TWindowsObject-Elementfunktion 405
HasVScrollBar
  TScroller-Datenelement 380
Hauptfenster 21, 30
  anpassen 35
  Anwendungsobjekte und 103
  erstellen 30, 32, 320
  Größe 123, 305
  Größe verändern 106
  Initialisierung 103f.
  MDI-Anwendungen und 199
  Menüs und 59
  schließen 106, 320, 399
  Titel 107
hbrBackground
  WNDCLASS-Element 129
hCursor
  WNDCLASS-Element 127, 129
Header-Dateien 291
  ObjectWindows 25
hIcon
  WNDCLASS-Element 128
HideList
  TComboBox-Elementfunktion 165, 330
Hierarchie
  Streams 411
  von ObjectWindows 316
Hilfesystem
  Windows 305
hInstance
  TModule-Datenelement 370
Hintergrundfarbe
  Bildschirmkontext 263
  Fenster 129, 392
Hot Links 254, 257
hPrevInstance
  TApplication-Datenelement 319
HScroll
  TScroller-Elementfunktion 383
HWindow 34
  TWindowsObject-Datenelement 396

I
ID_DLIST-Konstante 349
ID_FIRST-Konstante 80, 118, 338
ID_FLIST-Konstante 349
ID_FNAME-Konstante 350
ID_OK-Konstante 187
ID_XXXX-Konstanten 433
IDC_ARROW-Konstante 129
IDC_BEAM-Konstante 127
IDC_CROSS-Konstante 129
IDC_IBEAM-Konstante 129
IDC_WAIT-Konstante 129
IDI_ Konstanten 296
IDI_APPLICATION-Konstante 128
IdleAction
  TApplication-Elementfunktion 320
IDOK-Konstante 338
IDs
  Fenster 332
  Menüauswahlen 294
  Menüs 57
  Ressourcen 57
  Steuerelemente 79, 338
  Stringtabellen 296
  untergeordnete Fenster 118
ifpstream-Klasse 414
Index
  Listenposition 359
Init
  pstream-Elementfunktion 423
  TComboBox-Konstruktor 164
InitApplication
  TApplication-Element-
  funktion 107, 320

InitChild
TMDIFrame-Element-funktion 199, 368
InitClientWindow
TMDIFrame-Element-funktion 199, 369
InitInstance
TApplication-Element-funktion 106, 294, 320
InitMainWindow
MDI und 199
TApplication-Element-funktion 31, 105, 320
initTypes
pstream-Elementfunktion 423
Insert
TEdit-Elementfunktion 173, 344
InsertString
TListBox-Elementfunktion 159, 361
INSTALL 3
Installation 3
InstanceHashValue
TScroller-Datenelement 380
Instanzen-Initialisierung 319
Instanzenbindung 31
InvalidateRect-Funktion 44
iofstream 205
iopstream 205
iopstream-Klase 415
ipstream 205
ipstream-Klasse 415
Friends 417
isA
Object-Elementfunktion 317
TApplication-Elementfunktion 321
TDialog-Elementfunktion 338
TModule-Elementfunktion 372
TScroller-Elementfunktion 383
TWindow-Elementfunktion 392
TWindowsObject-Elementfunktion 405
isAssociation
Object-Elementfunktion 317
IsClipboardFormatAvailable 247
IsDefPB
TButton-Datenelement 323
isEqual
Object-Elementfunktion 317
TModule-Elementfunktion 372
TScroller-Elementfunktion 383
TWindowsObject-Elementfunktion 405
IsExhausted
SafetyPool-Elementfunktion 436
IsFlagSet
TWindowsObject-Elementfunktion 405
IsModal
TDialog-Datenelement 334
IsModified
TEdit-Elementfunktion 172, 344
IsNewFile
TFileWindow-Datenelement 351
IsReplaceOp
TEditWindow-Datenelement 346
isSortable
Object-Elementfunktion 317
IsVisibleRect
TScroller-Elementfunktion 135, 383
Iterator-Elementfunktionen
untergeordnete Fenster 114

K
Kapselung 316
KBHandlerWnd
TApplication-Datenelement 319
KERNEL.EXE 19
Klassen
Attribute 295
Fenster- 126ff.
Fensterregistrierung 392
gemeinsam genutzte 237ff.
MDI-Registrierung 368
Registrierung 126
Ressourcen und 295
Standard-ObjectWindows 313
Stile 128
stream-fähige 411ff.
Klassenbezeichner
Object 317
TApplication 321
Klassenbezeichnungsstring
TApplication 321
TWindowsObject und 406
Kollektionen
Streams und 212
Kombinationsfenster 162, 328ff.
ändern 165
ausklappende 163
ausklappende Listen- 164
Beispiel 165
Botschaften und 329
einfache 163
erstellen 164
Listen und 329
Listen vs. 162
Listenbereich 165

Stile 165, 328
Transferpuffer 185
Konstanten
Stil- 92
Konstruktor
Schnittstellenobjekte 112
stream-fähiger Klassen 209
TApplication 103
TCheckBox 175
TComboBox 164
TDialog 142
TEdit 168
TFileWindow 139
TGroupBox 178
TListBox 158
TRadioButton 175
TScrollBar 180
TScroller 131
TStatic 166
TWindow 59
TWindowsObject 111
Konventionen
typographische 6
Konversation (Datenaustausch) 250ff.
aufbauen 251
beenden 253
siehe auch Dynamischer Datenaustausch
Koordinatensystem
Abbildungsmodus und 263
Bildschirmkontext und 263
Kopier-Konstruktor
Objekt 316

L
lastThat
Object-Elementfunktion 317
lbColor
LOGBRUSH-Feld 267
LBN_-Botschaften 161
LBN_SELCHANGE-Botschaft 162
LBS_NOTIFY-Stil 359
LBS_OWNERDRAW-Stil 193
LBS_SORT-Stil 358
LBS_STANDARD-Stil 358
lbStyle
LOGBRUSH-Feld 266
LibMain-Funktion 234
Rückgabewert 234
LineMagnitude
TScrollBar-Datenelement 180, 375
LineTo-Funktion 47, 276
Linien
zeichnen 276
Linien zeichnen 264
Linienstile 265
__link-Makro 433
Liste untergeordneter Fenster 113
Listen 158, 358ff.
abfragen 160
ändern 159
Auswahl von 162
Beispiel 162
Benachrichtigungsbotschaften 161
Botschaften und 360
Einträge bearbeiten 359
Ereignisse und 161
Erstellung 158
sortieren 159
Stile 358
Strings und 159ff.
Transferpuffer 185
LMEM_DISCARDABLE-Flag 223, 225
LMEM_FIXED-Flag 223
LMEM_MODIFY-Flag 225
LMEM_MOVEABLE-Flag 223, 225
LMEM_NOCOMPACT-Flag 223, 225
LMEM_NODISCARD-Flag 223, 225
LMEM_ZEROINIT-Flag 225
LoadAccelerator-Funktion 294
LoadString-Funktion 297
LocalAlloc-Funktion 223
LocalCompact-Funktion 226
LocalDiscard-Funktion 224
LocalFlags-Funktion 225
LocalFree-Funktion 224
LocalLock-Funktion 223
LocalReAlloc-Funktion 224f.
LocalSize-Funktion 223, 225
LocalUnlock-Funktion 223
LOGBRUSH-Record 264
LOGFONT-Record 264
LOGPEN-Record 264
lokaler Heap
siehe lokaler Speicher
lokaler Speicher 221ff.
fester 223
Flags 223ff.
freigeben 224
Größe 223
Handles und 223
Maximalgröße 222f.
mit Attribut »discardable« 226
nicht verschiebbarer 224

reservieren 223
sperren 223
verwerfen 224
lopnColor
LOGPEN-Feld 265
lopnStyle
LOGPEN-Feld 265
lopnWidth
LOGPEN-Feld 265
LowMemory
TModule-Elementfunktion 372
LowMemory-Funktion 307
LParam
Botschaftsparameter 116
lpCmdLine
TModule-Datenelement 370
lpszMenuName
WNDCLASS-Element 129

M
MainWindow
TApplication-Datenelement 31, 319
MakeWindow 143
Create und 125
Create vs. 65, 307
Error und 308
nicht-modaler Dialog und 143
Rückgabewerte 65
TApplication-Elementfunktion 65, 308
TModule-Elementfunktion 372
TWindowObject-Elementfunktion 125
Mausbewegungen 45
Mausklicks (Koordinaten für) 43
Mauszeiger 295
ändern 126
Symbole vs. 295
vordefinierte 129
voreingestellter 338
MDI
Client-Fenster 363
Tastenkürzelbotschaften 321
untergeordnete Fenster 364ff.
Mehrfachvererbung 318
Menu
TWindowAttr-Feld 59
Menü-Ressourcen 293
Menübefehle
Datei-Dialoge für 350
Menüpunkte
Editierfelder und 170
Menüs 57ff.
Befehle beantworten 294
Bitmaps und 300
Botschaften und 60
Einträge 294, 346
Ereignisse 57
Hauptfenster und 59
Hauptmenüpunkt 58
IDs 57
laden 59, 293
Objekte und 59
Ressourcen 57
Standard- 129, 304
Stringtabellen und 297
Unterpunkte 58
MessageLoop
TApplication-Element-
funktion 104, 108f., 321
Module 319, 370ff.
ObjectWindows und 74
Objekte 103
Status 370
moduleClass-Konstante 433
MoveTo-Funktion 47, 276
Multidokumentschnittstelle (MDI) 197
Beispiel 202
Botschaften und 200
Client-Fenster 197
Fenster erzeugen 198
Komponenten 197
Menüs und 201
Objekte 90, 198
Rahmenfenster 198

N
Name
TModule-Datenelement 370
Namenskonventionen 85
nameOf
Object-Elementfunktion 317
TApplication-Elementfunktion 321
TComboBox-Elementfunktion 330
TDialog-Elementfunktion 338
TModule-Elementfunktion 372
TScroller-Elementfunktion 383
TStatic-Elementfunktion 388
TWindow-Elementfunktion 392
TWindowsObject-Elementfunktion 406
nCmdShow
TApplication-Datenelement 319
nebengeordnete Fenster 405
in Streams schreiben 406
new
Object-Elementfunktion 317

new-Operator 316, 434
NewFile
  TFileWindow-Elementfunktion 352
Next
  TWindowsObject-Elementfunktion 406
NF_XXXX-Konstanten 434
nicht-modale Dialgofenster
  Gebrauch 144
  Hauptfenster vs. 144
NotifyParent
  TGroupBox-Datenelement 355

O
Object-Klasse 316
ObjectWindows 13
  DOS-Hilfe 5
  Header-Dateien 25
  Hierarchie 86
  IDE und 27
  Installieren 3
  Klassenbibliothek 24
  Kommandozeilen-Tools 27, 29
  Komponenten 427
  Konventionen 85
  Make-Datei und 27
  Online-Hilfe 4
  Projektdatei und 27
  Standardklassen 313
Objekte
  Anwendungen 30, 86, 103ff.
  Bildlaufleisten 90
  Dialogfenster 88
  Fenster- 33, 88
  fortbestehende 203
  Modul 86
  Multidokumentschnittstelle 90
  Schnittstellen- 13, 34, 87
  Steuerelemente 89
obm_Close-Bitmap 298
obm_DnArrow-Bitmap 298
obm_Zoom-Bitmap 298
ODADrawEntire
  TControl-Elementfunktion 332
ODAFocus
  TControl-Elementfunktion 333
ODASelect
  TControl-Elementfunktion 333
ofpstream-Klasse 418
OK
  TDialog-Elementfunktion 144, 338
Open
  fpbase-Elementfunktion 412
  fpstream-Elementfunktion 413
  ifpstream-Elementfunktion 414
  ofpstream-Elementfunktion 419
  TFileWindow-Elementfunktion 352
OpenClipboard-Funktion 246f.
operator ! ()
  pstream-Elementfunktion 423
Operator < 318
  Friend-Funktion von Object 317
  opstream-Friends 421
  überschreiben für Streams 205
Operator > 318
  ipstream-Friends 417
  überschreiben für Streams 205
operator void *()
  pstream-Elementfunktion 423
Operatoren
  new 316
  verknüpfen 205
opstream 205
opstream-Klasse 419
  Friends 421

P
P_id_type 435
PageMagnitude
  TScrollBar-Datenelement 180, 375
Paint
  Bildlauf in Fenstern und 131
  TWindow-Element-
  funktion 52, 272ff., 393
Paletten 262, 281 ff.
  -Botschaften 284
  ändern 283
  erstellen 282
  Informationen über 283
  zeichnen mit 282
Parent
  TWindows-Datenelement 34
  TWindowsObject-Datenelement 396
Paste
  TEdit-Elementfunktion 344
PathName
  TFileDialog-Datenelement 348
PC_RESERVED-Konstante 284
PCchar-Typ 432
Pchar-Typ 432
PCvoid-Typ 432
Pie-Funktion 279
Pinsel
  Arten 263
  Bildschirmkontext und 264

Bitmaps als 266, 298
Farben 267
Schraffurmuster 267
virtuelle 266
Polygon-Funktion 280
PolyLine-Funktion 276
Position
Zeichen in Editierfeld 340
Positionszeiger
verschieben 378
PostMessage-Funktion 250f., 253
Ppstream-Klasse
Friends 424
Previous
TWindowsObject-Elementfunktion 406
printOn
Object-Elementfunktion 317
TModule-Elementfunktion 373
TScroller-Elementfunktion 383
TWindowsObject-Elementfunktion 406
Probleme mit Borland C++ 7
ProcessAccels
TApplication-Elementfunktion
108, 321
ProcessAppMsg
TApplication-Elementfunktion 321
ProcessDlgMessage
TApplication-Elementfunktion 108
ProcessDlgMsg
TApplication-Elementfunktion 321
ProcessMDIAccels
TApplication-Element-
funktion 108, 322
Prompt
TInputDialog-Datenelement 357
Protected Mode
Speicherverwaltung und 221
pstream-Klasse 422
Puffer
Eingabe-Dialoge und 357
Textlänge 386
Zeichen- 328
PutChildPtr
TWindowsObject-Element-
funktion 406
PutChildren
TWindowsObject-Element-
funktion 406
PutSiblingPtr
TWindowsObject-Element-
funktion 406
Pvoid-Typ 432

R
Rahmenfenster 365ff.
MDI 197
.RC-Dateien 289
rdbuf
fpbase-Elementfunktion 413
fpstream-Elementfunktion 413
ifpstream-Elementfunktion 414
ofpstream-Elementfunktion 419
pstream-Elementfunktion 423
rdstate
pstream-Elementfunktion 423
read
in stream-fähige Klassen linken 211
TCheckBox-Elementfunktion 327
TComboBox-Elementfunktion 330
TDialog-Elementfunktion 338
TEditWindow-Elementfunktion 347
TFileWindow-Elementfunktion 353
TGroupBox-Elementfunktion 356
TInputDialog-Elementfunktion 357
TMDIClient-Elementfunktion 364
TMDIFrame-Elementfunktion 369
TScrollBar-Elementfunktion 377
TScroller-Elementfunktion 383
TStatic-Elementfunktion 388
TStreamable-Elementfunktion 424
TWindow-Elementfunktion 392
TWindowsObject-Elementfunktion 407
read-Elementfunktion 208, 215
readByte
ipstream-Elementfunktion 416
readBytes
ipstream-Elementfunktion 416
readData
ipstream-Elementfunktion 416
README-Datei 4
readPrefix
ipstream-Elementfunktion 417
readString
ipstream-Elementfunktion 417
readSuffix
ipstream-Elementfunktion 417
readWord
ipstream-Elementfunktion 417
Real Mode
Speicherverwaltung und 221
RealizePalette-Elementfunktion 282f.
Rectangle-Funktion 278
Register
TWindowsObject-Element-
funktion 407

RegisterClipboardFormat-Funktion 245
registerObject
  ipstream-Elementfunktion 417
  opstream-Elementfunktion 420
RegisterWindowMessage-Funktion 250
Registrierung
  Fensterklassen 126
Registrierungsattribute 127
  ändern 126
  Hintergrundfarbe 129
  Mauszeiger 129
  Standardmenü 129
  Stile 128
  Symbol 128
ReleaseCapture-Funktion 47
RemoveClient
  TWindowsObject-Elementfunktion 407
ReplaceWith
  TFileWindow-Elementfunktion 353
.RES-Datei 288
ResetSelections
  TListBoxData-Elementfunktion 440
Ressourcen 287
  Arten 287
  Bitmaps 297
  Dateien 57
  Dialogfenster und 141, 295
  entfernen 287
  erzeugen 57, 287
  Header-Dateien und 291
  hinzufügen 288
  ID 57
  laden 288, 293
  Mauszeiger 295
  Menüs 57, 293
  Namen 293
  Standardmenü 129
  Steuerelemente und 154
  Strings 296
Ressourcencompiler 288
  Kommandozeilenoptionen 289
  Script-Datei 289
Ressourcendateien
  erstellen 26
RestoreMemory
  TModule-Elementfunktion 373
Result
  Botschaftsparameter 116
  TMessage-Element 116
RGB-Makro 266
Rint-Typ 432
RoundRect-Funktion 278
RPvoid-Typ 432
RTMessage-Typ 43, 116, 432
Run
  TApplication-Element-
  funktion 104, 108, 322

S
safetyPool
  SafetyPool-Datenelement 435
Save
  TFileWindow-Elementfunktion 353
SaveAs
  TFileWindow-Elementfunktion 353
SB_BOTTOM-Konstante 377
SB_LINEDOWN-Konstante 377
SB_LINEUP-Konstante 377
SB_PAGEDOWN-Konstante 377
SB_PAGEUP-Konstante 378
SB_THUMBPOSITION-Konstante
  378, 383f.
SB_THUMBTRACK-Konstante
  378, 383f.
SB_TOP-Konstante 378
SBBottom
  TScrollBar-Elementfunktion 183, 377
SBLineDown
  TScrollBar-Elementfunktion 183, 377
SBLineUp
  TScrollBar-Elementfunktion 183, 377
SBPageDown
  TScrollBar-Elementfunktion 183, 377
SBPageUp
  TScrollBar-Elementfunktion 183, 378
SBS_HORZ-Stil 180, 376
SBS_VERT-Stil 180, 376
SBThumbPosition
  TScrollBar-Elementfunktion 183, 378
SBThumbTrack
  TScrollBar-Elementfunktion 183, 378
SBTop
  TScrollBar-Elementfunktion 183, 378
Schaltfelder 175ff., 373ff.
  Beispiel 178
  erzeugen 175
  Stile 176
  Transferpuffer 186
  siehe auch Wahlfelder
Schnittstellenelemente
  anzeigen 112
  entfernen 112
  erzeugen 112
  Fehler 308

gleichzeitige Erstellung 401
Verhalten 395
verknüpfen 65
Schnittstellenobjekte 13, 34, 87, 111ff.
Fenster 122
Konstruktor 112
Steuerelemente 145
Zweck 111
Schraffurmuster 267
Schrift
Bildschirmkontext und 261
virtuelle 267
Schriftartenin den Handbüchern 6
Script-Datei 289
Scroll
TEdit-Elementfunktion 173, 345
ScrollBy
TScroller-Elementfunktion 134, 383
Scroller
TWindow-Datenelement 389
Scroller-Attribute 130
scrollerClass-Konstante 436
ScrollTo
TScroller-Elementfunktion 134, 384
SD_FILEOPEN-Ressourcen-
Bezeichner 349
SD_FILESAVE-Ressourcen-
Bezeichner 349
SD_INPUTDIALOG-Konstante 357
SD_REPLACE-Konstante 386
SD_SEARCH-Konstante 386
SD_XXXX-Konstanten 437
Search
TEdit-Elementfunktion 345
SearchStruct
TEditWindow-Datenelement 346
seekg
ipstream-Elementfunktion 417
seekp
opstream-Elementfunktion 420
SelCount
TListBoxData-Datenelement 439
SelectFileName
TFileDialog-Elementfunktion 350
Selection
TComboBoxData-Datenelement 438
SelectionChanged
TGroupBox-Elementfunktion 356
SelectObject-Funktion 50
SelectPalette-Elementfunktion 282
SelectSelection
TEdit-Elementfunktion 173
SelectString
TListBoxData-Elementfunktion 440
SelStrings
TListBoxData-Datenelement 440
SendDlgItemMsg
TDialog-Elementfunktion 146, 338
SendDlgItemsMsg
TDialog-Elementfunktion 99
SendDlgItemsMsg-Funktion 99
SendMessage-Funktion 99, 251f.
Server 250
setbuf
fpbase-Elementfunktion 413
SetCaption
TDialog-Elementfunktion 338
TWindowsObject-Elementfunktion 407
SetCapture-Funktion 47
SetCheck
TCheckBox-Elementfunktion 177, 327
SetClipboardData-Funktion 246
SetEditSel
TComboBox-Elementfunktion 330
SetFileName
TFileWindow-Elementfunktion 353
SetFlags
TWindowsObject-Elementfunktion 407
SetKBHandler
TApplication-Elementfunktion 322
SetPageSize
TScroller-Elementfunktion 135, 384
SetPaletteEntries-Funktion 283
SetParent
TWindowsObject-Elementfunktion 407
SetPosition
TScrollBar-Elementfunktion 181, 378
SetRange
TScrollBar-Elementfunktion 180f., 378
TScroller-Elementfunktion 134, 384
SetSBarRange
TScroller-Elementfunktion 384
SetSelection
TEdit-Elementfunktion 345
SetSelIndex
TListBox-Elementfunktion 162, 361
SetSelIndexes
TListBox-Elementfunktion 361
SetSelString
TListBox-Elementfunktion 162, 362
SetSelStrings
TListBox-Elementfunktion 362
setstate
pstream-Elementfunktion 424

SetText
  TComboBox-Elementfunktion 330
  TEdit-Elementfunktion 173
  TStatic-Elementfunktion 167, 388
SetTransferBuffer
  TWindowsObject-Elementfunktion 407
SetUnits
  TScroller-Elementfunktion 384
SetupWindow
  TButton-Elementfunktion 324
  TComboBox-Elementfunktion 330
  TDialog-Elementfunktion 339
  TEdit-Elementfunktion 345
  TFileDialog-Elementfunktion 350
  TFileWindow-Elementfunktion 353
  TInputDialog-Elementfunktion 357
  TMDIFrame-Elementfunktion 199f., 369
  TScrollBar-Elementfunktion 378
  TWindow-Elementfunktion 79, 393
  TWindowsObject-Elementfunktion 408
Show
  TWindowsObject-Element-funktion 112, 408
ShowList
  TComboBox-Elementfunktion 165, 330
ShutDownWindow
  TDialog-Elementfunktion 339
  TWindowsObject-Elementfunktion 408
Sicherheitsbereich
  Größe 307
  Implementierung 435
  new-Operator und 434
  TModule und 372
Size
  SafetyPool-Datenelement 436
smart callbacks 95
Software 3
Speicher
  gemeinsam genutzter 231
  globaler 226
  große Blöcke 222
  Großverbraucher 309
  lokal vs. global 222
  mit dem Attribut »discardable« 220
  sichere Zuweisung 307
  Sicherheitsbereich 306
  Speicherknappheit 230
  Sperre aufheben 221
  sperren 221
  Sperrenzähler 223
  verwerfbarer 220
  zusammenfassen 220ff.
  siehe auch lokaler Speicher
  globaler Speicher
Speicherreservierung
  Editierfelder und 343
  Fehlerkonstanten 431
  für Objekte 316
  Gültigkeit des Objekts 373
  Sicherheitsbereich 435
Speicherverwaltung 219ff.
  Handles und 221
  LRU-Algorithmus 230
  Multitasking 219
  Protected Mode und 221
  Real Mode und 221
Sperrenzähler 223
  überprüfen 226
SS_LEFT-Stil 387
SS_NOPREFIX-Stil 167
SS_SIMPLE-Stil 167
Standard-Botschaftsbearbeitung 99, 118
  Fenster 396
state
  pstream-Datenelement 422
statische Steuerelemente 165, 386 ff.
  ändern 167
  erzeugen 166
  Informationen über 167
  löschen 167
  Stile 166
  Transferpuffer 185
  unterstreichen 166
Status
  Fehlermeldung 309
  TModule-Datenelement 370
  TWindowsObject-Datenelement 396
  TWindowsObject-Elementfunktion 308
  von Auswahlfeldern 325
Steuerelemente 77, 153ff., 331ff.
  Aktionsschalter 174
  als abhängige untergeordnete Fenster 76
  als Fenster 77
  als untergeordnete Fenster 145
  Benachrichtigungsbotschaften 147, 156
  benutzergezeichnete 332
  Botschaften senden an 146
  Botschaften und 99, 145, 156
  Botschaftsbeantwortung 80
  Daten übertragen 184
  Dialogfenster und 145
  entfernen 155

Ereignisse und 80
Fenster und 73, 154
freigeben 155
IDs 79, 338
manipulieren 156
Objekte 89, 154
Ressourcen und 77, 154
statische 386ff.
verwalten 78
voreingestellte modifizieren 191
zeichenbar machen 192
zeichnen 333
Steuerelementobjekte
anwenderdefinierte 191
Anzeige 80
Dialogfenster und 146
erstellen 186
Stifte
ändern 50
Bildschirmkontext und 261
erzeugen 51
Linienstile 265
virtuelle 265
Stile 92
Aktionsschalter 174
Bildlaufleisten 179f.
Editierfelder 168
Erstellungsstil ändern 191
Fenster 122ff., 159
Klassen 128
Kombinationsfenster 165
kombinieren 92
Linien 265
Listen 159
statische Steuerelemente 166
virtuelle Pinsel 266
Stream
lesen aus 209
stream-fähige Klassen 203, 411ff.
build-Elementfunktion 209
BUILDER-Typ 425
erstellen 203, 211
ipstream und 205
Konstruktoren 209
lesen 415
read-Elementfunktion 207
Registrierung 211, 425
schreiben 420
TWindowsObject und 203
verwenden 212
write-Elementfunktion 207
stream-fähige Objekte
lesen und schreiben 414ff.
read-Elementfunktion 72
write-Elementfunktion 72
Stream-Verwaltung 207
Code linken 207ff.
TStreamable und 207
TStreamableClass und 207
TStreamableTypes und 207
StreamableInit-Typ 437
streamableName
streamfähige Klassen und 210, 214
TStreamable-Elementfunktion 424
Streams 203ff.
Ein-/Ausgabeoperationen 412
Hierarchie 411
Kollektionen und 212
schreiben in 209
Strings
aus Streams lesen 416
in Array einfügen 438
in Streams schreiben 420
Listen und 359ff.
TComboBoxData-Datenelement 438
TListBoxData-Datenelement 440
Stringtabellen
Dialogfenter und 296
IDs 296
laden 296
Menüs und 297
Style
TWindowsAttr-Datenelement 124
WNDCLASS-Element 128
Suchen
in Listen 359
TSearchDialog und 385
Symbole
Fenster 126
Mauszeiger vs. 295
MDI-Fenster und 367
vordefinierte 128, 296
System-Thema 260
Systemanforderungen 3

T
TActionFunc-Typ 437
TActionMemFunc-Typ 437
TApplication
Datenelemente 21
hInstance-Datenelement 21
hPrevInstance-Datenelement 21
Klassenbezeichner 427
lpCmdLine-Datenelement 21

nCmdShow-Datenelement 21
Status-Datenelement 31
Voraussetzungen 30
TApplication-Klasse 86, 103, 318f.
Tastatureingaben
aktives Fenster 319
Bearbeitungsmechanismus 322
Tastenkürzel
Befehle und 294
Botschaftsbearbeitung 321
definierte 57
laden 106, 294
MDI-Botschaften und 108
TButton-Klasse 322
TButton-Konstruktor 174
TCheckBox-Elementfunktion 175
TCheckBox-Klasse 175, 325
TComboBox 163
TComboBox-Klasse 328
TComboBoxData-Klasse 437
TCondFunc-Typ 438
TCondMemFunc-Typ 438
TControl-Klasse 77, 331
TDialog
Klassenbezeichner 431
TDialog-Klasse 334
TDialog-Konstruktor 142
TDialogAttr-Typ 439
Technische Beratung 7
TEdit-Klasse 340
TEdit-Konstruktor 168
TEditWindow-Fensterklasse 345
TEditWindow-Klasse 136
tellg
ipstream-Elementfunktion 417
tellp
opstream-Elementfunktion 420
Text
editieren 136
Editierfelder und 341ff.
in Editerfeld auswählen 350
in statischen Steuerelementen 386
Länge in Kombinationsfenstern 328
suchen 347
zeichnen 42
TextLen
TComboBox-Datenelement 328
TStatic-Datenelement 386
TextOut-Funktion 43, 275
Textpuffer
Länge 386
TF_GETDATA-Konstante 188, 327, 331, 358, 362, 379, 388, 409
TF_SETDATA-Konstante 188, 327, 331, 358, 362, 379, 388, 409
TF_SIZEDATA-Konstante 327, 331, 362, 379, 388, 409
TF_XXXX-Transfer-Funktion-Konstanten 439
TFileDialog-Klasse 151, 348
TFileDialog-Konstruktoren 151
TFileWindow-Klasse 136, 350
TGroupbox-Elementfunktion 178
TGroupBox-Klasse 354
TGroupBox-Konstruktor 178
THELP 5
TileChildren
TMDIClient-Elementfunktion 364
TMDIFrame-Elementfunktion 201, 369
TInputDialog-Klasse 356
Titel 320, 353
Fenster 346, 389
Hauptfenster 107
Title
TWindowsObject-Datenelement 396
TListBox-Klasse 158, 358
TListBoxData-Klasse 439
TMDIClient-Klasse 363
TMDIFrame-Datenelement 198
TMDIFrame-Klasse 365
TMessage-Struktur 36
TMessage-Typ 43, 440
TModule-Klasse 86, 103, 370
Toggle
TCheckBox-Elementfunktion 177, 327
TPointArray-Klasse 213
TrackMode
TScroller-Datenelement 380
TRadioButton-Elementfunktion 175
TRadioButton-Klasse 175, 373
Transfer
Rückgabewert 188
TCheckBox-Elementfunktion 327
TComboBox-Elementfunktion 331
TControl-Elementfunktion 188
TListBox-Elementfunktion 362
TScrollBar-Elementfunktion 379
TStatic-Elementfunktion 388
TWindowsObject-Elementfunktion 408
TransferBuffer
TDialog-Datenelement 186
TWindow-Datenelement 186
TWindowsObject-Datenelement 397

TransferData
  TInputDialog-Elementfunktion 357
  TWindowsObject-Element-
  funktion 188, 408
Transferdaten
  Größe 327
Transferpuffer
  aktualisieren 187
  Auswahlfelder 186
  Beispiel 188
  Bildlaufleisten 186, 379
  definieren 184
  Editierfelder und 185, 346
  Fenster und 408
  Kombinationsfenster und 185, 331, 437
  Listen und 182, 362, 439
  Schaltfelder 186
  statische Steuerelemente und 185, 388
  TObjectWindow und 397
TScrollbar-Klasse 375
TScrollBar-Konstruktor 180
TScrollBarData-Typ 441
TScroller
  Attribute 130
  Bereich 130
  Einheiten 130
  Fenster und 131
  Klassenbezeichner 436
  Seitenlänge 130
  Zeilenlänge 130
TScroller-Klasse 129, 379
TScroller-Objekt 130
TSearchDialog-Klasse 385
TSearchStruct-Typ 441
TStatic-Klasse 386
TStatic-Konstruktor 166
TStreamable 71
  Stream-Verwaltung und 207
TStreamable-Klasse 424
  Friends 425
TStreamableClass
  Stream-Verwaltung und 207
TStreamableClass-Klasse 425
  __DELTA-Makro 426
  Friends 426
TStreamableTypes
  Stream-Verwaltung und 207
TWindow 32f.
  Objekte 88
TWindow-Klasse 121, 388
TWindowAttr 389
TWindowAttr-Struktur 122
TWindowAttr-Typ 442
TWindowsObject 88
TWindowsObject-Klasse 111, 395
Typenprüfung 316
types
  pstream-Datenelement 422
typographische Konventionen 6

U
übergeordnete Fenster 34, 63, 113
  Vorfahre vs. 64
Uncheck
  TCheckBox-Elementfunktion 176, 327
Undo
  TEdit-Elementfunktion 170, 345
untergeordnete Fenster 34, 63, 113
  als Datenelemente 78
  Attribute setzen 124
  Destruktor 114
  Dialogfenster als 142
  erstellen 113, 336, 399
  ID-Bereich 118
  in Stream schreiben 406
  konstruieren 113
  Liste mit 113
  MDI 197, 199, 201
  selbständige 63
  unselbständige 64
  Zugehörigkeitsverhältnis 64
UpdateColors-Funktion 284
UpdateFileName
  TFileDialog-Elementfunktion 350
UpdateListBoxes
  TFileDialog-Elementfunktion 350
USER.EXE 19

V
ValidWindow
  TModule-Elementfunktion 373
virtuell (Destruktor) 316
virtuelle Pinsel 266
virtuelle Schriften 267
VScroll
  TScroller-Elementfunktion 384

W
Wahlfelder 175
  erzeugen 175
  Informationen über 176
  Status ändern 176
WB_AUTOCREATE-Konstante 399
WB_FROMRESOURCE-Konstante 391ff.

WB_XXXX-Konstanten 442
WEP-Funktion 234
  Rückgabewert 234
Window
  TScroller-Datenelement 380
windowClass-Konstante 443
Windows
  API 14, 91
  Funktionen 91, 93
  ObjectWindows und 91
  Paletten-Manager 281
  Struktur 18
  Zwischenablage 245
Windows-Botschaften 95
WM_ACTIVATE-Botschaft 409
WM_CLOSE-Botschaft 109, 339
WM_COMMAND-Botschaft 117
WM_COMPACTING-Botschaft 230
WM_DDE_ACK-Botschaft 252, 255
WM_DDE_ADVISE-Botschaft 254
WM_DDE_DATA-Botschaft 254f.
WM_DDE_EXECUTE-Botschaft 254
WM_DDE_INITIATE-Botschaft 251f.
WM_DDE_POKE-Botschaft 254, 258
WM_DDE_REQUEST-Botschaft 254f.
WM_DDE_TERMINATE-Botschaft 251ff.
WM_DDE_UNADVISE-Botschaft 254
WM_DESTROY-Botschaft 409
WM_DEXTROYCLIPBOARD-
  Botschaft 249
WM_DRAWITEM-Botschaft 333, 409
WM_HSCROLL-Botschaft 182, 409
WM_LBUTTONDOWN-Botschaft 35
WM_LBUTTONUP-Botschaft 45
WM_MDIACTIVATE-Botschaft 201
WM_MOUSEMOVE-Botschaft 45
WM_NCDESTROY-Botschaft 410
WM_PAINT-Botschaft 394
WM_PALETTECHANGED-Botschaft 284
WM_QUERYENDSESSION-
  Botschaft 339
WM_RBUTTONDOWN-Botschaft 35
WM_RENDERALLFORMATS-
  Botschaft 249
WM_VSCROLL-Botschaft 182, 410
WM_XXXX-Konstanten 443
WMActivate
  TMDIFrame-Elementfunktion 369
  TWindow-Elementfunktion 393
  TWindowsObject-Elementfunktion 409
WMClose
  TDialog-Elementfunktion 339
  TWindowsObject-Elementfunktion 409
WMCommand
  TWindowsObject-Elementfunktion 409
WMCreate
  TWindow-Elementfunktion 393
WMDestroy
  TWindowsObject-Elementfunktion 409
WMDrawItem
  TControl-Elementfunktion 333
  TWindowsObject-Elementfunktion 409
WMGetDlgCode
  TButton-Elementfunktion 324
WMHScroll
  TWindow-Elementfunktion 393
  TWindowsObject-Element-
  funktion 409
WMInitDialog
  TDialog-Elementfunktion 339
WMLButtonDown
  TWindow-Elementfunktion 394
WMMDIActivate
  TMDIClient-Elementfunktion 364
  TWindow-Elementfunktion 394
WMMove
  TWindow-Elementfunktion 394
WMNCDestroy
  TWindowsObject-Elementfunktion 410
WMPaint
  TControl-Elementfunktion 333
  TMDIClient-Elementfunktion 365
  TWindow-Elementfunktion 394
WMQueryEndSession
  TDialog-Elementfunktion 339
  TWindowsObject-Elementfunktion 410
WMSetFocus
  TEditWindow-Element-
  funktion 137, 347
WMSize
  TEditWindow-Elementfunktion 347
  TWindow-Elementfunktion 395
WMVScroll
  TWindow-Elementfunktion 395
  TWindowsObject-Elementfunktion 410
WNDCLASS
  Standardwerte 128
WNDCLASS-Struktur 128
WParam
  Botschaftsparameter 116
write
  in stream-fähige Klassen linken 211
  TCheckBox-Elementfunktion 327
  TComboBox-Elementfunktion 331

TDialog-Elementfunktion 340
TEditWindow-Elementfunktion 348
TFileWindow-Elementfunktion 353f.
TGroupBox-Elementfunktion 356
TInputDialog-Elementfunktion 358
TMDIClient-Elementfunktion 365
TMDIFrame-Elementfunktion 369
TScrollBar-Elementfunktion 379
TScroller-Elementfunktion 384
TStatic-Elementfunktion 388
TStreamable-Elementfunktion 425
TWindow-Elementfunktion 395
TWindowsObject-Elementfunktion 410
writeByte
opstream-Elementfunktion 420
writeBytes
opstream-Elementfunktion 420
writeData
opstream-Elementfunktion 420
writePrefix
opstream-Elementfunktion 420
writeString
opstream-Elementfunktion 421
writeSuffix
opstream-Elementfunktion 421
writeWord
opstream-Elementfunktion 421
WS_AUTOCHECKBOX-Stil 326, 329
WS_AUTORADIOBUTTON-Stil 374
WS_BORDER-Stil 168, 358
WS_CHILD-Stil 165, 326, 329, 332, 363, 374, 387
WS_CLIPCHILDREN-Stil 363
WS_CLIPSIBLINGS-Stil 390
WS_GROUP-Stil 363, 387
WS_HSCROLL-Konstante 131
WS_HSCROLL-Stil 169, 179, 375
WS_OVERLAPPEDWINDOW-Stil 390
WS_TABSTOP-Stil 168, 326, 329, 332, 355, 363
WS_VISIBLE-Stil 112, 165, 168, 326, 329, 332, 363, 374, 387, 390
WS_VSCROLL-Konstante 131
WS_VSCROLL-Stil 169, 179, 358, 375

X
XLine
TScroller-Datenelement 380
XPage
TScroller-Datenelement 380
XPos
TScroller-Datenelement 381
XRange
TScroller-Datenelement 381
XRangeValue
TScroller-Elementfunktion 385
XScrollValue
TScroller-Elementfunktion 385
XUnit
TScroller-Datenelement 130, 381

Y
YLine (TScroller-Datenelement) 381
YPage (TScroller-Datenelement) 381
YPos (TScroller-Datenelement) 381
YRange (TScroller-Datenelement) 381
YRangeValue
(TScroller-Elementfunktion) 385
YScrollValue
(TScroller-Elementfunktio)n 385
YUnit
(TScroller-Datenelement) 130, 381

Z
Zeichen
Editierfelder und 340f.
Puffer 328
Zeichenfunktionen (GDI) 275
Zeichenwerkzeuge 48, 262ff.
Attribute 264
auswählen 50
Handles 48
Standardsatz 49
verwenden 273
virtuelle 265
vordefinierte 264
zeichnen und 52
Zeichnen
benutzergezeichnetes Steuerelement 332
Bildschirmkontext und 52
geometrische Formen 278
Text 42
Zeichenwerkzeuge und 52
ZERO (Object-Datenelement) 316
Zugriffsattribute
private/protected/public 315
Zusammenfassung von Speicher 220
Zwischenablage 245ff.
Daten laden aus 245, 247
Editierfelder und 170
Formate 245
Speichern von Daten in 246
Text kopieren in 342

PROGRAMMIER-
HANDBUCH

OBJECT WINDOWS®
C++

BORLAND

BORLAND GMBH, GAUTINGER STR. 10,POSTFACH 1209, D-8130 STARNBERG, TELEFON 0 81 51 / 263-0, TELEFAX 0 81 51 / 142 09
CORPORATE HEADQUARTERS: 1800 GREEN HILLS ROAD, P.O. BOX 660001, SCOTTS VALLEY, CA 95067-0001. (408) 438-530
OFFICES IN: AUSTRALIA, DENMARK, FRANCE, GERMANY, ITALY, JAPAN, NEW ZEALAND, SINGAPORE, SWEDEN AND THE UNITED KINDOM * ARTIKEL-NR. 135301